



SymPy Documentation

Release 1.2

SymPy Development Team

July 09, 2018

CONTENTS

1	Installation	1
1.1	Anaconda	1
1.2	Git	1
1.3	Other Methods	2
1.4	Run SymPy	2
1.5	Mpmath	2
1.6	Questions	2
2	SymPy Tutorial	3
2.1	Preliminaries	3
2.2	Introduction	4
2.3	Gotchas	8
2.4	Basic Operations	13
2.5	Printing	16
2.6	Simplification	23
2.7	Calculus	37
2.8	Solvers	44
2.9	Matrices	48
2.10	Advanced Expression Manipulation	55
3	Gotchas and Pitfalls	65
3.1	Introduction	65
3.2	Equals Signs (=)	65
3.3	Variables	66
3.4	Symbolic Expressions	69
3.5	Special Symbols	74
3.6	Getting help from within SymPy	76
4	SymPy User's Guide	81
4.1	Introduction	81
4.2	Learning SymPy	81
4.3	SymPy's Architecture	83
4.4	Contributing	90
5	SymPy Modules Reference	91
5.1	SymPy Core	91
5.2	Combinatorics Module	202
5.3	Number Theory	298
5.4	Basic Cryptography Module	335
5.5	Concrete Mathematics	360

5.6	Numerical evaluation	374
5.7	Structural Details of Code Generation with SymPy	380
5.8	Numeric Computation	423
5.9	Functions Module	425
5.10	Geometry Module	557
5.11	Holonomic Functions	657
5.12	Symbolic Integrals	666
5.13	Numeric Integrals	694
5.14	Integration over Polytopes	700
5.15	Lie Algebra Module	703
5.16	Logic Module	717
5.17	Matrices	729
5.18	Polynomials Manipulation Module	785
5.19	Printing System	1019
5.20	Plotting Module	1059
5.21	Pyglet Plotting Module	1072
5.22	Assumptions module	1076
5.23	Term rewriting	1102
5.24	Series Module	1105
5.25	Sets	1132
5.26	Simplify	1154
5.27	Details on the Hypergeometric Function Expansion Module	1180
5.28	Stats	1189
5.29	ODE	1230
5.30	PDE	1287
5.31	Solvers	1295
5.32	Diophantine	1314
5.33	Inequality Solvers	1339
5.34	Solveset	1342
5.35	Tensor Module	1360
5.36	Utilities	1390
5.37	Parsing input	1455
5.38	Calculus	1460
5.39	Physics Module	1469
5.40	Category Theory Module	1736
5.41	Differential Geometry Module	1753
5.42	Vector Module	1768
5.43	Contributions to docs	1807
6	SymPy Special Topics	1809
6.1	Introduction	1809
6.2	Finite Difference Approximations to Derivatives	1809
7	Development Tips: Comparisons in Python	1815
7.1	Introduction	1815
7.2	Hashing	1815
7.3	Method Resolution	1816
7.4	General Notes and Caveats	1818
7.5	Script To Verify This Guide	1818
8	Wiki	1823
8.1	FAQ	1823
9	SymPy Papers	1825

10 Planet SymPy	1827
11 SymPy logos	1829
12 Blogs, News, Magazines	1831
13 About	1833
13.1 SymPy Development Team	1833
13.2 Financial and Infrastructure Support	1846
13.3 License	1847
14 Citing SymPy	1851
Bibliography	1853
Python Module Index	1875
Index	1879

INSTALLATION

The SymPy CAS can be installed on virtually any computer with Python 2.7 or above. SymPy does require `mpmath` Python library to be installed first. The recommended method of installation is through Anaconda, which includes `mpmath`, as well as several other useful libraries. Alternatively, some Linux distributions have SymPy packages available.

SymPy officially supports Python 2.7, 3.4, 3.5, 3.6 and PyPy.

1.1 Anaconda

Anaconda is a free Python distribution from Continuum Analytics that includes SymPy, Matplotlib, IPython, NumPy, and many more useful packages for scientific computing. This is recommended because many nice features of SymPy are only enabled when certain libraries are installed. For example, without Matplotlib, only simple text-based plotting is enabled. With the IPython notebook or qtconsole, you can get nicer $\text{\LaTeX}$ printing by running `init_printing()`.

If you already have Anaconda and want to update SymPy to the latest version, use:

```
conda update sympy
```

1.2 Git

If you wish to contribute to SymPy or like to get the latest updates as they come, install SymPy from git. To download the repository, execute the following from the command line:

```
git clone git://github.com/sympy/sympy.git
```

To update to the latest version, go into your repository and execute:

```
git pull origin master
```

If you want to install SymPy, but still want to use the git version, you can run from your repository:

```
python setupegg.py develop
```

This will cause the installed version to always point to the version in the git directory.

1.3 Other Methods

Various Linux distributions have SymPy available as a package. You may also install SymPy from source or using pip.

1.4 Run SymPy

After installation, it is best to verify that your freshly-installed SymPy works. To do this, start up Python and import the SymPy libraries:

```
$ python
>>> from sympy import *
```

From here, execute some simple SymPy statements like the ones below:

```
>>> x = Symbol('x')
>>> limit(sin(x)/x, x, 0)
1
>>> integrate(1/x, x)
log(x)
```

For a starter guide on using SymPy effectively, refer to the [SymPy Tutorial](#) (page 3).

1.5 Mpmath

Versions of SymPy prior to 1.0 included [mpmath](#), but it now depends on it as an external dependency. If you installed SymPy with Anaconda, it will already include mpmath. Use:

```
conda install mpmath
```

to ensure that it is installed.

If you do not wish to use Anaconda, you can use `pip install mpmath`.

If you use `mpmath` via `sympy.mpmath` in your code, you will need to change this to use just `mpmath`. If you depend on code that does this that you cannot easily change, you can work around it by doing:

```
import sys
import mpmath
sys.modules['sympy.mpmath'] = mpmath
```

before the code that imports `sympy.mpmath`. It is recommended to change code that uses `sympy.mpmath` to use `mpmath` directly wherever possible.

1.6 Questions

If you have a question about installation or SymPy in general, feel free to visit our chat on [Gitter](#). In addition, our [mailing list](#) is an excellent source of community support.

If you think there's a bug or you would like to request a feature, please open an [issue ticket](#).

SYMPY TUTORIAL

2.1 Preliminaries

This tutorial assumes that the reader already knows the basics of the Python programming language. If you do not, the [official Python tutorial](#) is excellent.

This tutorial assumes a decent mathematical background. Most examples require knowledge lower than a calculus level, and some require knowledge at a calculus level. Some of the advanced features require more than this. If you come across a section that uses some mathematical function you are not familiar with, you can probably skip over it, or replace it with a similar one that you are more familiar with. Or look up the function on Wikipedia and learn something new. Some important mathematical concepts that are not common knowledge will be introduced as necessary.

2.1.1 Installation

Quick Tip

You do not need to install SymPy to try it. You can use the online shell at <http://live.sympy.org>, or the shell at the bottom right of this documentation page.

You will need to install SymPy first. See the [installation guide](#) (page 1).

Alternately, you can just use the SymPy Live Sphinx extension to run the code blocks in the browser. For example, click on the green “Run code block in SymPy Live” button below

```
>>> from sympy import *
>>> x = symbols('x')
>>> a = Integral(cos(x)*exp(x), x)
>>> Eq(a, a.doit())
Eq(Integral(exp(x)*cos(x), x), exp(x)*sin(x)/2 + exp(x)*cos(x)/2)
```

The SymPy Live shell in the bottom corner will pop up and evaluate the code block. You can also click any individual line to evaluate it one at a time.

The SymPy Live shell is a fully interactive Python shell. You can type any expression in the input box to evaluate it. Feel free to use it throughout the tutorial to experiment.

To show or hide the SymPy Live shell at any time, just click the green button on the bottom right of the screen.

By default, the SymPy Live shell uses $\text{\LaTeX}$ for output. If you want the output to look more like the output in the documentation, change the output format to `Str` or `Unicode` in the settings.

If you wish to modify an example before evaluating it, change the evaluation mode to “copy” in the SymPy Live settings. This will cause clicking on an example to copy the example to the SymPy Live shell, but not evaluate it, allowing you to change it before execution. You can also use the up/down arrow keys on your keyboard in the input box to move through the shell history.

The SymPy Live shell is also available at <http://live.sympy.org>, with extra features, like a mobile phone enhanced version and saved history.

2.1.2 Exercises

This tutorial was the basis for a tutorial given at the 2013 SciPy conference in Austin, TX. The website for that tutorial is [here](#). It has links to videos, materials, and IPython notebook exercises. The IPython notebook exercises in particular are recommended to anyone going through this tutorial.

2.1.3 About This Tutorial

This tutorial aims to give an introduction to SymPy for someone who has not used the library before. Many features of SymPy will be introduced in this tutorial, but they will not be exhaustive. In fact, virtually every functionality shown in this tutorial will have more options or capabilities than what will be shown. The rest of the SymPy documentation serves as API documentation, which extensively lists every feature and option of each function.

These are the goals of this tutorial:

- To give a guide, suitable for someone who has never used SymPy (but who has used Python and knows the necessary mathematics).
- To be written in a narrative format, which is both easy and fun to follow. It should read like a book.
- To give insightful examples and exercises, to help the reader learn and to make it entertaining to work through.
- To introduce concepts in a logical order.
- To use good practices and idioms, and avoid antipatterns. Functions or methodologies that tend to lead to antipatterns are avoided. Features that are only useful to advanced users are not shown.
- To be consistent. If there are multiple ways to do it, only the best way is shown.
- To avoid unnecessary duplication, it is assumed that previous sections of the tutorial have already been read.

Feedback on this tutorial, or on SymPy in general is always welcome. Just write to our [mailing list](#).

2.2 Introduction

2.2.1 What is Symbolic Computation?

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

Let's take an example. Say we wanted to use the built-in Python functions to compute square roots. We might do something like this

```
>>> import math
>>> math.sqrt(9)
3.0
```

9 is a perfect square, so we got the exact answer, 3. But suppose we computed the square root of a number that isn't a perfect square

```
>>> math.sqrt(8)
2.82842712475
```

Here we got an approximate result. 2.82842712475 is not the exact square root of 8 (indeed, the actual square root of 8 cannot be represented by a finite decimal, since it is an irrational number). If all we cared about was the decimal form of the square root of 8, we would be done.

But suppose we want to go further. Recall that $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$. We would have a hard time deducing this from the above result. This is where symbolic computation comes in. With a symbolic computation system like SymPy, square roots of numbers that are not perfect squares are left unevaluated by default

```
>>> import sympy
>>> sympy.sqrt(3)
sqrt(3)
```

Furthermore—and this is where we start to see the real power of symbolic computation—symbolic results can be symbolically simplified.

```
>>> sympy.sqrt(8)
2*sqrt(2)
```

2.2.2 A More Interesting Example

The above example starts to show how we can manipulate irrational numbers exactly using SymPy. But it is much more powerful than that. Symbolic computation systems (which by the way, are also often called computer algebra systems, or just CASs) such as SymPy are capable of computing symbolic expressions with variables.

As we will see later, in SymPy, variables are defined using `symbols`. Unlike many symbolic manipulation systems, variables in SymPy must be defined before they are used (the reason for this will be discussed in the [next section](#) (page 8)).

Let us define a symbolic expression, representing the mathematical expression $x + 2y$.

```
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> expr = x + 2*y
>>> expr
x + 2*y
```

Note that we wrote `x + 2*y` just as we would if `x` and `y` were ordinary Python variables. But in this case, instead of evaluating to something, the expression remains as just `x + 2*y`. Now let us play around with it:

```
>>> expr + 1
x + 2*y + 1
>>> expr - x
2*y
```

Notice something in the above example. When we typed `expr - x`, we did not get `x + 2*y - x`, but rather just `2*y`. The `x` and the `-x` automatically canceled one another. This is similar to how `sqrt(8)` automatically turned into `2*sqrt(2)` above. This isn't always the case in SymPy, however:

```
>>> x*expr
x*(x + 2*y)
```

Here, we might have expected $x(x+2y)$ to transform into $x^2 + 2xy$, but instead we see that the expression was left alone. This is a common theme in SymPy. Aside from obvious simplifications like $x - x = 0$ and $\sqrt{8} = 2\sqrt{2}$, most simplifications are not performed automatically. This is because we might prefer the factored form $x(x+2y)$, or we might prefer the expanded form $x^2 + 2xy$. Both forms are useful in different circumstances. In SymPy, there are functions to go from one form to the other

```
>>> from sympy import expand, factor
>>> expanded_expr = expand(x*expr)
>>> expanded_expr
x**2 + 2*x*y
>>> factor(expanded_expr)
x*(x + 2*y)
```

2.2.3 The Power of Symbolic Computation

The real power of a symbolic computation system such as SymPy is the ability to do all sorts of computations symbolically. SymPy can simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, and much, much more, and do it all symbolically. It includes modules for plotting, printing (like 2D pretty printed output of math formulas, or $\LaTeX$), code generation, physics, statistics, combinatorics, number theory, geometry, logic, and more. Here is a small sampling of the sort of symbolic power SymPy is capable of, to whet your appetite.

```
>>> from sympy import *
>>> x, t, z, nu = symbols('x t z nu')
```

This will make all further examples pretty print with unicode characters.

```
>>> init_printing(use_unicode=True)
```

Take the derivative of $\sin(x)e^x$.

```
>>> diff(sin(x)*exp(x), x)
      x      x
e  ·sin(x) + e  ·cos(x)
```

Compute $\int (e^x \sin(x) + e^x \cos(x)) dx$.

```
>>> integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
      x
e  ·sin(x)
```

Compute $\int_{-\infty}^{\infty} \sin(x^2) dx$.

```
>>> integrate(sin(x**2), (x, -oo, oo))

$$\frac{\sqrt{2} \cdot \sqrt{\pi}}{2}$$

```

Find $\lim_{x \rightarrow 0} \frac{\sin(x)}{x}$.

```
>>> limit(sin(x)/x, x, 0)
1
```

Solve $x^2 - 2 = 0$.

```
>>> solve(x**2 - 2, x)
[-sqrt(2), sqrt(2)]
```

Solve the differential equation $y'' - y = e^t$.

```
>>> y = Function('y')
>>> dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t))

$$y(t) = C_2 \cdot e^{-t} + \left( C_1 + \frac{-1}{2} \right) \cdot e^t$$

```

Find the eigenvalues of $\begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$.

```
>>> Matrix([[1, 2], [2, 2]]).eigenvals()

$$\left\{ -\frac{3}{2} + \frac{\sqrt{17}}{2}: 1, -\frac{3}{2} + \frac{\sqrt{17}}{2}: 1 \right\}$$

```

Rewrite the Bessel function $J_\nu(z)$ in terms of the spherical Bessel function $j_\nu(z)$.

```
>>> besselj(nu, z).rewrite(jn)

$$\frac{\sqrt{2} \cdot \sqrt{z} \cdot j_n(\nu - 1/2, z)}{\sqrt{\pi}}$$

```

Print $\int_0^\pi \cos^2(x) dx$ using L^AT_EX.

```
>>> latex(Integral(cos(x)**2, (x, 0, pi)))
\int_{0}^{\pi} \cos^2{x} \left( x \right) dx
```

2.2.4 Why SymPy?

There are many computer algebra systems out there. [This](#) Wikipedia article lists many of them. What makes SymPy a better choice than the alternatives?

First off, SymPy is completely free. It is open source, and licensed under the liberal BSD license, so you can modify the source code and even sell it if you want to. This contrasts with popular commercial systems like Maple or Mathematica that cost hundreds of dollars in licenses.

Second, SymPy uses Python. Most computer algebra systems invent their own language. Not SymPy. SymPy is written entirely in Python, and is executed entirely in Python. This means that if you already know Python, it is much easier to get started with SymPy, because

you already know the syntax (and if you don't know Python, it is really easy to learn). We already know that Python is a well-designed, battle-tested language. The SymPy developers are confident in their abilities in writing mathematical software, but programming language design is a completely different thing. By reusing an existing language, we are able to focus on those things that matter: the mathematics.

Another computer algebra system, Sage also uses Python as its language. But Sage is large, with a download of over a gigabyte. An advantage of SymPy is that it is lightweight. In addition to being relatively small, it has no dependencies other than Python, so it can be used almost anywhere easily. Furthermore, the goals of Sage and the goals of SymPy are different. Sage aims to be a full featured system for mathematics, and aims to do so by compiling all the major open source mathematical systems together into one. When you call some function in Sage, such as `integrate`, it calls out to one of the open source packages that it includes. In fact, SymPy is included in Sage. SymPy on the other hand aims to be an independent system, with all the features implemented in SymPy itself.

A final important feature of SymPy is that it can be used as a library. Many computer algebra systems focus on being usable in interactive environments, but if you wish to automate or extend them, it is difficult to do. With SymPy, you can just as easily use it in an interactive Python environment or import it in your own Python application. SymPy also provides APIs to make it easy to extend it with your own custom functions.

2.3 Gotchas

To begin, we should make something about SymPy clear. SymPy is nothing more than a Python library, like NumPy, Django, or even modules in the Python standard library `sys` or `re`. What this means is that SymPy does not add anything to the Python language. Limitations that are inherent in the Python language are also inherent in SymPy. It also means that SymPy tries to use Python idioms whenever possible, making programming with SymPy easy for those already familiar with programming with Python. As a simple example, SymPy uses Python syntax to build expressions. Implicit multiplication (like `3x` or `3 x`) is not allowed in Python, and thus not allowed in SymPy. To multiply 3 and `x`, you must type `3*x` with the `*`.

2.3.1 Symbols

One consequence of this fact is that SymPy can be used in any environment where Python is available. We just import it, like we would any other library:

```
>>> from sympy import *
```

This imports all the functions and classes from SymPy into our interactive Python session. Now, suppose we start to do a computation.

```
>>> x + 1
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Oops! What happened here? We tried to use the variable `x`, but it tells us that `x` is not defined. In Python, variables have no meaning until they are defined. SymPy is no different. Unlike many symbolic manipulation systems you may have used, in SymPy, variables are not defined automatically. To define variables, we must use symbols.

```
>>> x = symbols('x')
>>> x + 1
x + 1
```

`symbols` takes a string of variable names separated by spaces or commas, and creates Symbols out of them. We can then assign these to variable names. Later, we will investigate some convenient ways we can work around this issue. For now, let us just define the most common variable names, `x`, `y`, and `z`, for use through the rest of this section

```
>>> x, y, z = symbols('x y z')
```

As a final note, we note that the name of a Symbol and the name of the variable it is assigned to need not have anything to do with one another.

```
>>> a, b = symbols('b a')
>>> a
b
>>> b
a
```

Here we have done the very confusing thing of assigning a Symbol with the name `a` to the variable `b`, and a Symbol of the name `b` to the variable `a`. Now the Python variable named `a` points to the SymPy Symbol named `b`, and visa versa. How confusing. We could have also done something like

```
>>> crazy = symbols('unrelated')
>>> crazy + 1
unrelated + 1
```

This also shows that Symbols can have names longer than one character if we want.

Usually, the best practice is to assign Symbols to Python variables of the same name, although there are exceptions: Symbol names can contain characters that are not allowed in Python variable names, or may just want to avoid typing long names by assigning Symbols with long names to single letter Python variables.

To avoid confusion, throughout this tutorial, Symbol names and Python variable names will always coincide. Furthermore, the word “Symbol” will refer to a SymPy Symbol and the word “variable” will refer to a Python variable.

Finally, let us be sure we understand the difference between SymPy Symbols and Python variables. Consider the following:

```
x = symbols('x')
expr = x + 1
x = 2
print(expr)
```

What do you think the output of this code will be? If you thought 3, you’re wrong. Let’s see what really happens

```
>>> x = symbols('x')
>>> expr = x + 1
>>> x = 2
>>> print(expr)
x + 1
```

Changing `x` to 2 had no effect on `expr`. This is because `x = 2` changes the Python variable `x` to 2, but has no effect on the SymPy Symbol `x`, which was what we used in creating `expr`.

When we created `expr`, the Python variable `x` was a Symbol. After we created, it, we changed the Python variable `x` to 2. But `expr` remains the same. This behavior is not unique to SymPy. All Python programs work this way: if a variable is changed, expressions that were already created with that variable do not change automatically. For example

```
>>> x = 'abc'
>>> expr = x + 'def'
>>> expr
'abcdef'
>>> x = 'ABC'
>>> expr
'abcdef'
```

Quick Tip

To change the value of a Symbol in an expression, use `subs`

```
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
3
```

In this example, if we want to know what `expr` is with the new value of `x`, we need to reevaluate the code that created `expr`, namely, `expr = x + 1`. This can be complicated if several lines created `expr`. One advantage of using a symbolic computation system like SymPy is that we can build a symbolic representation for `expr`, and then substitute `x` with values. The correct way to do this in SymPy is to use `subs`, which will be discussed in more detail later.

```
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
3
```

2.3.2 Equals signs

Another very important consequence of the fact that SymPy does not extend Python syntax is that `=` does not represent equality in SymPy. Rather it is Python variable assignment. This is hard-coded into the Python language, and SymPy makes no attempts to change that.

You may think, however, that `==`, which is used for equality testing in Python, is used for SymPy as equality. This is not quite correct either. Let us see what happens when we use `==`.

```
>>> x + 1 == 4
False
```

Instead of treating `x + 1 == 4` symbolically, we just got `False`. In SymPy, `==` represents exact structural equality testing. This means that `a == b` means that we are asking if $a = b$. We always get a `bool` as the result of `==`. There is a separate object, called `Eq`, which can be used to create symbolic equalities

```
>>> Eq(x + 1, 4)
Eq(x + 1, 4)
```

There is one additional caveat about `==` as well. Suppose we want to know if $(x+1)^2 = x^2+2x+1$. We might try something like this:


```
>>> (x + 1)**2 == x**2 + 2*x + 1
False
```

We got `False` again. However, $(x + 1)^2$ does equal $x^2 + 2x + 1$. What is going on here? Did we find a bug in SymPy, or is it just not powerful enough to recognize this basic algebraic fact?

Recall from above that `==` represents exact structural equality testing. “Exact” here means that two expressions will compare equal with `==` only if they are exactly equal structurally. Here, $(x + 1)^2$ and $x^2 + 2x + 1$ are not the same symbolically. One is the power of an addition of two terms, and the other is the addition of three terms.

It turns out that when using SymPy as a library, having `==` test for exact structural equality is far more useful than having it represent symbolic equality, or having it test for mathematical equality. However, as a new user, you will probably care more about the latter two. We have already seen an alternative to representing equalities symbolically, `Eq`. To test if two things are equal, it is best to recall the basic fact that if $a = b$, then $a - b = 0$. Thus, the best way to check if $a = b$ is to take $a - b$ and simplify it, and see if it goes to 0. We will learn [later](#) (page 23) that the function to do this is called `simplify`. This method is not infallible—in fact, it can be [theoretically proven](#) that it is impossible to determine if two symbolic expressions are identically equal in general—but for most common expressions, it works quite well.

```
>>> a = (x + 1)**2
>>> b = x**2 + 2*x + 1
>>> simplify(a - b)
0
>>> c = x**2 - 2*x + 1
>>> simplify(a - c)
4*x
```

There is also a method called `equals` that tests if two expressions are equal by evaluating them numerically at random points.

```
>>> a = cos(x)**2 - sin(x)**2
>>> b = cos(2*x)
>>> a.equals(b)
True
```

2.3.3 Two Final Notes: `^` and `/`

You may have noticed that we have been using `**` for exponentiation instead of the standard `^`. That’s because SymPy follows Python’s conventions. In Python, `^` represents logical exclusive or. SymPy follows this convention:

```
>>> True ^ False
True
>>> True ^ True
False
>>> x^y
Xor(x, y)
```

Finally, a small technical discussion on how SymPy works is in order. When you type something like `x + 1`, the SymPy Symbol `x` is added to the Python int `1`. Python’s operator rules then allow SymPy to tell Python that SymPy objects know how to be added to Python ints, and so `1` is automatically converted to the SymPy Integer object.

This sort of operator magic happens automatically behind the scenes, and you rarely need to even know that it is happening. However, there is one exception. Whenever you combine a SymPy object and a SymPy object, or a SymPy object and a Python object, you get a SymPy object, but whenever you combine two Python objects, SymPy never comes into play, and so you get a Python object.

```
>>> type(Integer(1) + 1)
<class 'sympy.core.numbers.Integer'>
>>> type(1 + 1)
<... 'int'>
```

Note: On running the example above in SymPy Live, $(1+1)$ is wrapped by Integer, so it does not show the correct output.

This is usually not a big deal. Python ints work much the same as SymPy Integers, but there is one important exception: division. In SymPy, the division of two Integers gives a Rational:

```
>>> Integer(1)/Integer(3)
1/3
>>> type(Integer(1)/Integer(3))
<class 'sympy.core.numbers.Rational'>
```

But in Python `/` represents either integer division or floating point division, depending on whether you are in Python 2 or Python 3, and depending on whether or not you have run `from __future__ import division`:

```
>>> from __future__ import division
>>> 1/2
0.5
```

Note: On running the example above in SymPy Live, $(1/2)$ is wrapped by Integer, so it does not show the correct output.

To avoid this, we can construct the rational object explicitly

```
>>> Rational(1, 2)
1/2
```

This problem also comes up whenever we have a larger symbolic expression with `int/int` in it. For example:

```
>>> x + 1/2
x + 0.5
```

Note: On running the example above in SymPy Live, $(1/2)$ is wrapped by Integer, so it does not show the correct output.

This happens because Python first evaluates $1/2$ into 0.5 , and then that is cast into a SymPy type when it is added to `x`. Again, we can get around this by explicitly creating a Rational:

```
>>> x + Rational(1, 2)
x + 1/2
```

There are several tips on avoiding this situation in the [Gotchas and Pitfalls](#) (page 65) document.

2.3.4 Further Reading

For more discussion on the topics covered in this section, see [Gotchas and Pitfalls](#) (page 65).

2.4 Basic Operations

Here we discuss some of the most basic operations needed for expression manipulation in SymPy. Some more advanced operations will be discussed later in the [advanced expression manipulation](#) (page 55) section.

```
>>> from sympy import *
>>> x, y, z = symbols("x y z")
```

2.4.1 Substitution

One of the most common things you might want to do with a mathematical expression is substitution. Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method. For example

```
>>> expr = cos(x) + 1
>>> expr.subs(x, y)
cos(y) + 1
```

Substitution is usually done for one of two reasons:

1. Evaluating an expression at a point. For example, if our expression is $\cos(x) + 1$ and we want to evaluate it at the point $x = 0$, so that we get $\cos(0) + 1$, which is 2.

```
>>> expr.subs(x, 0)
2
```

2. Replacing a subexpression with another subexpression. There are two reasons we might want to do this. The first is if we are trying to build an expression that has some symmetry, such as x^{x^x} . To build this, we might start with $x^{**}y$, and replace y with $x^{**}y$. We would then get $x^{**}(x^{**}y)$. If we replaced y in this new expression with $x^{**}x$, we would get $x^{**}(x^{**}(x^{**}x))$, the desired expression.

```
>>> expr = x**y
>>> expr
x**y
>>> expr = expr.subs(y, x**y)
>>> expr
x**(x**y)
>>> expr = expr.subs(y, x**x)
>>> expr
x**(x**(x**x))
```

The second is if we want to perform a very controlled simplification, or perhaps a simplification that SymPy is otherwise unable to do. For example, say we have $\sin(2x) + \cos(2x)$,

and we want to replace $\sin(2x)$ with $2\sin(x)\cos(x)$. As we will learn later, the function `expand_trig` does this. However, this function will also expand $\cos(2x)$, which we may not want. While there are ways to perform such precise simplification, and we will learn some of them in the [advanced expression manipulation](#) (page 55) section, an easy way is to just replace $\sin(2x)$ with $2\sin(x)\cos(x)$.

```
>>> expr = sin(2*x) + cos(2*x)
>>> expand_trig(expr)
2*sin(x)*cos(x) + 2*cos(x)**2 - 1
>>> expr.subs(sin(2*x), 2*sin(x)*cos(x))
2*sin(x)*cos(x) + cos(2*x)
```

There are two important things to note about `subs`. First, it returns a new expression. SymPy objects are immutable. That means that `subs` does not modify it in-place. For example

```
>>> expr = cos(x)
>>> expr.subs(x, 0)
1
>>> expr
cos(x)
>>> x
x
```

Quick Tip

SymPy expressions are immutable. No function will change them in-place.

Here, we see that performing `expr.subs(x, 0)` leaves `expr` unchanged. In fact, since SymPy expressions are immutable, no function will change them in-place. All functions will return new expressions.

To perform multiple substitutions at once, pass a list of (old, new) pairs to `subs`.

```
>>> expr = x**3 + 4*x*y - z
>>> expr.subs([(x, 2), (y, 4), (z, 0)])
40
```

It is often useful to combine this with a list comprehension to do a large set of similar replacements all at once. For example, say we had $x^4 - 4x^3 + 4x^2 - 2x + 3$ and we wanted to replace all instances of x that have an even power with y , to get $y^4 - 4x^3 + 4y^2 - 2x + 3$.

```
>>> expr = x**4 - 4*x**3 + 4*x**2 - 2*x + 3
>>> replacements = [(x**i, y**i) for i in range(5) if i % 2 == 0]
>>> expr.subs(replacements)
-4*x**3 - 2*x + y**4 + 4*y**2 + 3
```

2.4.2 Converting Strings to SymPy Expressions

The `sympify` function (that's `sympify`, not to be confused with `simplify`) can be used to convert strings into SymPy expressions.

For example

```
>>> str_expr = "x**2 + 3*x - 1/2"
>>> expr = sympify(str_expr)
>>> expr
x**2 + 3*x - 1/2
>>> expr.subs(x, 2)
19/2
```

Warning: `sympify` uses `eval`. Don't use it on unsanitized input.

2.4.3 evalf

To evaluate a numerical expression into a floating point number, use `evalf`.

```
>>> expr = sqrt(8)
>>> expr.evalf()
2.82842712474619
```

SymPy can evaluate floating point expressions to arbitrary precision. By default, 15 digits of precision are used, but you can pass any number as the argument to `evalf`. Let's compute the first 100 digits of π .

```
>>> pi.evalf(100)
3.
↪ 14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706
```

To numerically evaluate an expression with a Symbol at a point, we might use `subs` followed by `evalf`, but it is more efficient and numerically stable to pass the substitution to `evalf` using the `subs` flag, which takes a dictionary of Symbol: point pairs.

```
>>> expr = cos(2*x)
>>> expr.evalf(subs={x: 2.4})
0.0874989834394464
```

Sometimes there are roundoff errors smaller than the desired precision that remain after an expression is evaluated. Such numbers can be removed at the user's discretion by setting the `chop` flag to `True`.

```
>>> one = cos(1)**2 + sin(1)**2
>>> (one - 1).evalf()
-0.e-124
>>> (one - 1).evalf(chop=True)
0
```

2.4.4 lambdify

`subs` and `evalf` are good if you want to do simple evaluation, but if you intend to evaluate an expression at many points, there are more efficient ways. For example, if you wanted to evaluate an expression at a thousand points, using SymPy would be far slower than it needs to be, especially if you only care about machine precision. Instead, you should use libraries like [NumPy](#) and [SciPy](#).

The easiest way to convert a SymPy expression to an expression that can be numerically evaluated is to use the `lambdify` function. `lambdify` acts like a `lambda` function, except it converts the SymPy names to the names of the given numerical library, usually NumPy. For example

```
>>> import numpy
>>> a = numpy.arange(10)
>>> expr = sin(x)
>>> f = lambdify(x, expr, "numpy")
>>> f(a)
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
 -0.2794155   0.6569866   0.98935825  0.41211849]
```

Warning: `lambdify` uses `eval`. Don't use it on unsanitized input.

You can use other libraries than NumPy. For example, to use the standard library `math` module, use `"math"`.

```
>>> f = lambdify(x, expr, "math")
>>> f(0.1)
0.0998334166468
```

To use `lambdify` with numerical libraries that it does not know about, pass a dictionary of `sympy_name:numerical_function` pairs. For example

```
>>> def mysin(x):
...     """
...     My sine. Note that this is only accurate for small x.
...     """
...     return x
>>> f = lambdify(x, expr, {"sin":mysin})
>>> f(0.1)
0.1
```

2.5 Printing

As we have already seen, SymPy can pretty print its output using Unicode characters. This is a short introduction to the most common printing options available in SymPy.

2.5.1 Printers

There are several printers available in SymPy. The most common ones are

- `str`
- `srepr`
- ASCII pretty printer
- Unicode pretty printer
- LaTeX
- MathML

- Dot

In addition to these, there are also “printers” that can output SymPy objects to code, such as C, Fortran, Javascript, Theano, and Python. These are not discussed in this tutorial.

2.5.2 Setting up Pretty Printing

If all you want is the best pretty printing, use the `init_printing()` function. This will automatically enable the best printer available in your environment.

```
>>> from sympy import init_printing
>>> init_printing()
```

Quick Tip

You can also change the printer used in SymPy Live. Just change the “Output Format” in the settings.

If you plan to work in an interactive calculator-type session, the `init_session()` function will automatically import everything in SymPy, create some common Symbols, setup plotting, and run `init_printing()`.

```
>>> from sympy import init_session
>>> init_session()
```

Python console for SymPy 0.7.3 (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:

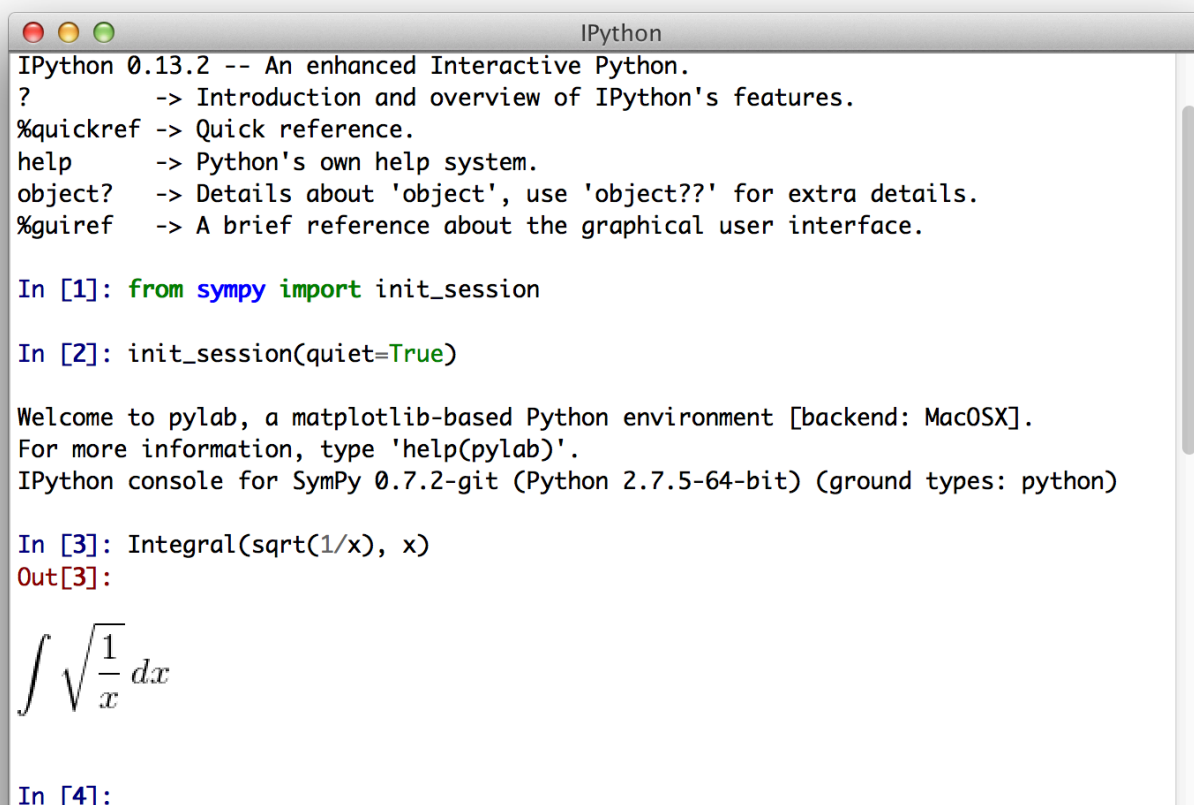
```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> init_printing() # doctest: +SKIP
```

Documentation can be found at <http://www.sympy.org>

```
>>>
```

In any case, this is what will happen:

- In the IPython QtConsole, if $\text{\LaTeX}$ is installed, it will enable a printer that uses $\text{\LaTeX}$.



```
IPython 0.13.2 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
%gui?   -> A brief reference about the graphical user interface.

In [1]: from sympy import init_session

In [2]: init_session(quiet=True)

Welcome to pylab, a matplotlib-based Python environment [backend: MacOSX].
For more information, type 'help(pylab)'.
IPython console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: python)

In [3]: Integral(sqrt(1/x), x)
Out[3]:

$$\int \sqrt{\frac{1}{x}} dx$$


In [4]:
```

If $\text{\LaTeX}$ is not installed, but Matplotlib is installed, it will use the Matplotlib rendering engine. If Matplotlib is not installed, it uses the Unicode pretty printer.

- In the IPython notebook, it will use MathJax to render $\text{\LaTeX}$.

```
In [1]: from sympy import *
x, y, z = symbols('x y z')
init_printing()
```

```
In [2]: Integral(sqrt(1/x), x)
```

```
Out[2]:

$$\int \sqrt{\frac{1}{x}} dx$$

```

- In an IPython console session, or a regular Python session, it will use the Unicode pretty printer if the terminal supports Unicode.


```

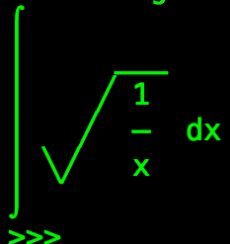
Python 2.7.5 (default, May 16 2013, 18:48:51)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import init_session
>>> init_session()
Python console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Documentation can be found at http://www.sympy.org

>>> Integral(sqrt(1/x), x)

```



```

>>>

```

- In a terminal that does not support Unicode, the ASCII pretty printer is used.

```

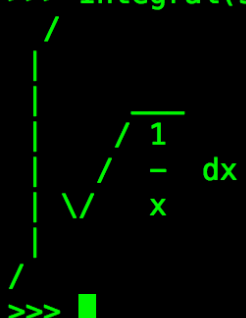
Python 2.7.5 (default, May 16 2013, 18:48:51)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import init_session
>>> init_session()
Python console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Documentation can be found at http://www.sympy.org

>>> Integral(sqrt(1/x), x)

```



```

>>>

```

To explicitly not use $\text{\LaTeX}$, pass `use_latex=False` to `init_printing()` or `init_session()`.
 To explicitly not use Unicode, pass `use_unicode=False`.

2.5.3 Printing Functions

In addition to automatic printing, you can explicitly use any one of the printers by calling the appropriate function.

str

To get a string form of an expression, use `str(expr)`. This is also the form that is produced by `print(expr)`. String forms are designed to be easy to read, but in a form that is correct Python syntax so that it can be copied and pasted. The `str()` form of an expression will usually look exactly the same as the expression as you would enter it.

```

>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> str(Integral(sqrt(1/x), x))
'Integral(sqrt(1/x), x)'
>>> print(Integral(sqrt(1/x), x))
Integral(sqrt(1/x), x)

```

```
>>> srepr(Integral(sqrt(1/x), x))
"Integral(Pow(Pow(Symbol('x'), Integer(-1)), Rational(1, 2)), Tuple(Symbol('x')))"
```

ASCII Pretty Printer

```
>>> pprint(Integral(sqrt(1/x), x), use_unicode=False)
```

$$\int \frac{1}{x} dx$$

```
>>> pretty(Integral(sqrt(1/x), x), use_unicode=False)
```

```
>>> print(pretty(Integral(sqrt(1/x), x), use_unicode=False))
```

$$\int \frac{1}{x} dx$$

```
>>> pprint(Integral(sqrt(1/x), x), use_unicode=True)
```

$$\frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

21

(continued from previous page)

$$\int \sqrt{\frac{1}{x}} dx$$

LaTeX

To get the LaTeX form of an expression, use `latex()`.

```
>>> print(latex(Integral(sqrt(1/x), x)))
\int \sqrt{\frac{1}{x}}\, dx
```

The `latex()` function has many options to change the formatting of different things. See [its documentation](#) (page 1045) for more details.

MathML

There is also a printer to MathML, called `print_mathml()`. It must be imported from `sympy.printing.mathml`.

```
>>> from sympy.printing.mathml import print_mathml
>>> print_mathml(Integral(sqrt(1/x), x))
<apply>
  <int/>
  <bvar>
    <ci>x</ci>
  </bvar>
  <apply>
    <root/>
    <apply>
      <power/>
      <ci>x</ci>
      <cn>-1</cn>
    </apply>
  </apply>
</apply>
```

`print_mathml()` prints the output. If you want the string, use the function `mathml()`.

Dot

The `dotprint()` function in `sympy.printing.dot` prints output to dot format, which can be rendered with Graphviz. See the [Advanced Expression Manipulation](#) (page 55) section for some examples of the output of this printer.

```
>>> from sympy.printing.dot import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{
# Graph style
"ordering"="out"
"rankdir"="TD"
#####
```

(continues on next page)

(continued from previous page)

```
# Nodes #
#####
"Add(Integer(2), Symbol(x))_()" ["color"="black", "label"="Add", "shape"="ellipse"];
"Integer(2)_()" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol(x)_()" ["color"="black", "label"="x", "shape"="ellipse"];
#####
# Edges #
#####
"Add(Integer(2), Symbol(x))_()" -> "Integer(2)_()";
"Add(Integer(2), Symbol(x))_()" -> "Symbol(x)_()";
}
```

2.6 Simplification

To make this document easier to read, we are going to enable pretty printing.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

2.6.1 simplify

Now let's jump in and do some interesting mathematics. One of the most useful features of a symbolic manipulation system is the ability to simplify mathematical expressions. SymPy has dozens of functions to perform various kinds of simplification. There is also one general function called `simplify()` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Here are some examples

```
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
>>> simplify(gamma(x)/gamma(x - 2))
(x - 2)*(x - 1)
```

Here, `gamma(x)` is $\Gamma(x)$, the [gamma function](#). We see that `simplify()` is capable of handling a large class of expressions.

But `simplify()` has a pitfall. It just applies all the major simplification operations in SymPy, and uses heuristics to determine the simplest result. But "simplest" is not a well-defined term. For example, say we wanted to "simplify" $x^2 + 2x + 1$ into $(x + 1)^2$:

```
>>> simplify(x**2 + 2*x + 1)
2
x  + 2·x + 1
```

We did not get what we want. There is a function to perform this simplification, called `factor()`, which will be discussed below.

Another pitfall to `simplify()` is that it can be unnecessarily slow, since it tries many kinds of simplifications before picking the best one. If you already know exactly what kind of simplification you are after, it is better to apply the specific simplification function(s) that apply those simplifications.

Applying specific simplification functions instead of `simplify()` also has the advantage that specific functions have certain guarantees about the form of their output. These will be discussed with each function below. For example, `factor()`, when called on a polynomial with rational coefficients, is guaranteed to factor the polynomial into irreducible factors. `simplify()` has no guarantees. It is entirely heuristical, and, as we saw above, it may even miss a possible type of simplification that SymPy is capable of doing.

`simplify()` is best when used interactively, when you just want to whittle down an expression to a simpler form. You may then choose to apply specific functions once you see what `simplify()` returns, to get a more precise result. It is also useful when you have no idea what form an expression will take, and you need a catchall function to simplify it.

2.6.2 Polynomial/Rational Function Simplification

`expand`

`expand()` is one of the most common simplification functions in SymPy. Although it has a lot of scopes, for now, we will consider its function in expanding polynomial expressions. For example:

```
>>> expand((x + 1)**2)
      2
x  + 2·x + 1
>>> expand((x + 2)*(x - 3))
      2
x  - x - 6
```

Given a polynomial, `expand()` will put it into a canonical form of a sum of monomials.

`expand()` may not sound like a simplification function. After all, by its very name, it makes expressions bigger, not smaller. Usually this is the case, but often an expression will become smaller upon calling `expand()` on it due to cancellation.

```
>>> expand((x + 1)*(x - 2) - (x - 1)*x)
-2
```

`factor`

`factor()` takes a polynomial and factors it into irreducible factors over the rational numbers. For example:

```
>>> factor(x**3 - x**2 + x - 1)
      2
(x - 1)·(x  + 1)
>>> factor(x**2*z + 4*x*y*z + 4*y**2*z)
      2
z·(x + 2·y)
```

For polynomials, `factor()` is the opposite of `expand()`. `factor()` uses a complete multivariate factorization algorithm over the rational numbers, which means that each of the factors returned by `factor()` is guaranteed to be irreducible.

If you are interested in the factors themselves, `factor_list` returns a more structured output.

```
>>> factor_list(x**2*z + 4*x*y*z + 4*y**2*z)
(1, [(z, 1), (x + 2*y, 2)])
```

Note that the input to `factor` and `expand` need not be polynomials in the strict sense. They will intelligently factor or expand any kind of expression (though note that the factors may not be irreducible if the input is no longer a polynomial over the rationals).

```
>>> expand((cos(x) + sin(x))**2)
      2      2
sin (x) + 2·sin(x)·cos(x) + cos (x)
>>> factor(cos(x)**2 + 2*cos(x)*sin(x) + sin(x)**2)
      2
(sin(x) + cos(x))
```

collect

`collect()` collects common powers of a term in an expression. For example

```
>>> expr = x*y + x - 3 + 2*x**2 - z*x**2 + x**3
>>> expr
      3      2      2
x  - x  ·z + 2·x  + x·y + x - 3
>>> collected_expr = collect(expr, x)
>>> collected_expr
      3      2
x  + x  ·(-z + 2) + x·(y + 1) - 3
```

`collect()` is particularly useful in conjunction with the `.coeff()` method. `expr.coeff(x, n)` gives the coefficient of x^n in `expr`:

```
>>> collected_expr.coeff(x, 2)
-z + 2
```

cancel

`cancel()` will take any rational function and put it into the standard canonical form, $\frac{p}{q}$, where p and q are expanded polynomials with no common factors, and the leading coefficients of p and q do not have denominators (i.e., are integers).

```
>>> cancel((x**2 + 2*x + 1)/(x**2 + x))
x + 1
-----
x
```

```
>>> expr = 1/x + (3*x/2 - 2)/(x - 4)
>>> expr
3·x
--- - 2
2      1
----- + -
x - 4    x
>>> cancel(expr)
      2
```

(continues on next page)

(continued from previous page)

$$\frac{3 \cdot x^2 - 2 \cdot x - 8}{2 \cdot x^2 - 8 \cdot x}$$

```
>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
```

$$\frac{x^2 y^2 - 2 x y z + x^2 z^2 + y^2 - 2 y z + z^2}{x^2 - 1}$$

```
>>> cancel(expr)
```

$$\frac{y^2 - 2 y z + z^2}{x - 1}$$

Note that since `factor()` will completely factorize both the numerator and the denominator of an expression, it can also be used to do the same thing:

```
>>> factor(expr)
```

$$\frac{(y - z)^2}{x - 1}$$

However, if you are only interested in making sure that the expression is in canceled form, `cancel()` is more efficient than `factor()`.

apart

`apart()` performs a [partial fraction decomposition](#) on a rational function.

```
>>> expr = (4*x**3 + 21*x**2 + 10*x + 12)/(x**4 + 5*x**3 + 5*x**2 + 4*x)
>>> expr
```

$$\frac{4 \cdot x^3 + 21 \cdot x^2 + 10 \cdot x + 12}{x^4 + 5 \cdot x^3 + 5 \cdot x^2 + 4 \cdot x}$$

```
>>> apart(expr)
```

$$\frac{2 \cdot x - 1}{x^2 + x + 1} - \frac{1}{x + 4} + \frac{3}{x}$$

2.6.3 Trigonometric Simplification

Note: SymPy follows Python's naming conventions for inverse trigonometric functions, which is to append an `a` to the front of the function's name. For example, the inverse cosine, or arc cosine, is called `acos()`.


```
>>> acos(x)
acos(x)
>>> cos(acos(x))
x
>>> asin(1)
 $\frac{\pi}{2}$ 
```

trigsimp

To simplify expressions using trigonometric identities, use `trigsimp()`.

```
>>> trigsimp(sin(x)**2 + cos(x)**2)
1
>>> trigsimp(sin(x)**4 - 2*cos(x)**2*sin(x)**2 + cos(x)**4)
cos(4*x)
-----
2
>>> trigsimp(sin(x)*tan(x)/sec(x))
sin(x)
```

`trigsimp()` also works with hyperbolic trig functions.

```
>>> trigsimp(cosh(x)**2 + sinh(x)**2)
cosh(2*x)
>>> trigsimp(sinh(x)/tanh(x))
cosh(x)
```

Much like `simplify()`, `trigsimp()` applies various trigonometric identities to the input expression, and then uses a heuristic to return the “best” one.

expand_trig

To expand trigonometric functions, that is, apply the sum or double angle identities, use `expand_trig()`.

```
>>> expand_trig(sin(x + y))
sin(x)*cos(y) + sin(y)*cos(x)
>>> expand_trig(tan(2*x))
2*tan(x)
-----
1 - tan(x)**2
```

Because `expand_trig()` tends to make trigonometric expressions larger, and `trigsimp()` tends to make them smaller, these identities can be applied in reverse using `trigsimp()`

```
>>> trigsimp(sin(x)*cos(y) + sin(y)*cos(x))
sin(x + y)
```

2.6.4 Powers

Before we introduce the power simplification functions, a mathematical discussion on the identities held by powers is in order. There are three kinds of identities satisfied by exponents

1. $x^a x^b = x^{a+b}$
2. $x^a y^a = (xy)^a$
3. $(x^a)^b = x^{ab}$

Identity 1 is always true.

Identity 2 is not always true. For example, if $x = y = -1$ and $a = \frac{1}{2}$, then $x^a y^a = \sqrt{-1}\sqrt{-1} = i \cdot i = -1$, whereas $(xy)^a = \sqrt{-1 \cdot -1} = \sqrt{1} = 1$. However, identity 2 is true at least if x and y are nonnegative and a is real (it may also be true under other conditions as well). A common consequence of the failure of identity 2 is that $\sqrt{x}\sqrt{y} \neq \sqrt{xy}$.

Identity 3 is not always true. For example, if $x = -1$, $a = 2$, and $b = \frac{1}{2}$, then $(x^a)^b = ((-1)^2)^{1/2} = \sqrt{1} = 1$ and $x^{ab} = (-1)^{2 \cdot 1/2} = (-1)^1 = -1$. However, identity 3 is true when b is an integer (again, it may also hold in other cases as well). Two common consequences of the failure of identity 3 are that $\sqrt{x^2} \neq x$ and that $\sqrt{\frac{1}{x}} \neq \frac{1}{\sqrt{x}}$.

To summarize

Identity	Sufficient conditions to hold	Counterexample when conditions are not met	Important consequences
1. $x^a x^b = x^{a+b}$	Always true	None	None
2. $x^a y^a = (xy)^a$	$x, y \geq 0$ and $a \in \mathbb{R}$	$(-1)^{1/2}(-1)^{1/2} \neq (-1 \cdot -1)^{1/2}$	$\sqrt{x}\sqrt{y} \neq \sqrt{xy}$ in general
3. $(x^a)^b = x^{ab}$	$b \in \mathbb{Z}$	$((-1)^2)^{1/2} \neq (-1)^{2 \cdot 1/2}$	$\sqrt{x^2} \neq x$ and $\sqrt{\frac{1}{x}} \neq \frac{1}{\sqrt{x}}$ in general

This is important to remember, because by default, SymPy will not perform simplifications if they are not true in general.

In order to make SymPy perform simplifications involving identities that are only true under certain assumptions, we need to put assumptions on our Symbols. We will undertake a full discussion of the assumptions system later, but for now, all we need to know are the following.

- By default, SymPy Symbols are assumed to be complex (elements of $\mathbb{C}$). That is, a simplification will not be applied to an expression with a given Symbol unless it holds for all complex numbers.
- Symbols can be given different assumptions by passing the assumption to `symbols()`. For the rest of this section, we will be assuming that x and y are positive, and that a and b are real. We will leave z , t , and c as arbitrary complex Symbols to demonstrate what happens in that case.

```
>>> x, y = symbols('x y', positive=True)
>>> a, b = symbols('a b', real=True)
>>> z, t, c = symbols('z t c')
```

Note: In SymPy, `sqrt(x)` is just a shortcut to `x**Rational(1, 2)`. They are exactly the same object.

```
>>> sqrt(x) == x**Rational(1, 2)
True
```

powsimp

`powsimp()` applies identities 1 and 2 from above, from left to right.

```
>>> powsimp(x**a*x**b)
a + b
x
>>> powsimp(x**a*y**a)
a
(x·y)
```

Notice that `powsimp()` refuses to do the simplification if it is not valid.

```
>>> powsimp(t**c*z**c)
c c
t · z
```

If you know that you want to apply this simplification, but you don't want to mess with assumptions, you can pass the `force=True` flag. This will force the simplification to take place, regardless of assumptions.

```
>>> powsimp(t**c*z**c, force=True)
c
(t·z)
```

Note that in some instances, in particular, when the exponents are integers or rational numbers, and identity 2 holds, it will be applied automatically.

```
>>> (z*t)**2
2 2
t · z
>>> sqrt(x*y)
√x·√y
```

This means that it will be impossible to undo this identity with `powsimp()`, because even if `powsimp()` were to put the bases together, they would be automatically split apart again.

```
>>> powsimp(z**2*t**2)
2 2
t · z
>>> powsimp(sqrt(x)*sqrt(y))
√x·√y
```

expand_power_exp / expand_power_base

`expand_power_exp()` and `expand_power_base()` apply identities 1 and 2 from right to left, respectively.

```
>>> expand_power_exp(x**(a + b))
a  b
x  · x
```

```
>>> expand_power_base((x*y)**a)
a  a
x  · y
```

As with `powsimp()`, identity 2 is not applied if it is not valid.

```
>>> expand_power_base((z*t)**c)
c
(t · z)
```

And as with `powsimp()`, you can force the expansion to happen without fiddling with assumptions by using `force=True`.

```
>>> expand_power_base((z*t)**c, force=True)
c  c
t  · z
```

As with identity 2, identity 1 is applied automatically if the power is a number, and hence cannot be undone with `expand_power_exp()`.

```
>>> x**2*x**3
5
x
>>> expand_power_exp(x**5)
5
x
```

powdenest

`powdenest()` applies identity 3, from left to right.

```
>>> powdenest((x**a)**b)
a · b
x
```

As before, the identity is not applied if it is not true under the given assumptions.

```
>>> powdenest((z**a)**b)
b
( a )
( z )
```

And as before, this can be manually overridden with `force=True`.

```
>>> powdenest((z**a)**b, force=True)
a · b
z
```

2.6.5 Exponentials and logarithms

Note: In SymPy, as in Python and most programming languages, `log` is the natural logarithm, also known as `ln`. SymPy automatically provides an alias `ln = log` in case you forget this.

```
>>> ln(x)
log(x)
```

Logarithms have similar issues as powers. There are two main identities

1. $\log(xy) = \log(x) + \log(y)$
2. $\log(x^n) = n \log(x)$

Neither identity is true for arbitrary complex x and y , due to the branch cut in the complex plane for the complex logarithm. However, sufficient conditions for the identities to hold are if x and y are positive and n is real.

```
>>> x, y = symbols('x y', positive=True)
>>> n = symbols('n', real=True)
```

As before, z and t will be Symbols with no additional assumptions.

Note that the identity $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$ is a special case of identities 1 and 2 by $\log\left(\frac{x}{y}\right) = \log\left(x \cdot \frac{1}{y}\right) = \log(x) + \log(y^{-1}) = \log(x) - \log(y)$, and thus it also holds if x and y are positive, but may not hold in general.

We also see that $\log(e^x) = x$ comes from $\log(e^x) = x \log(e) = x$, and thus holds when x is real (and it can be verified that it does not hold in general for arbitrary complex x , for example, $\log(e^{x+2\pi i}) = \log(e^x) = x \neq x + 2\pi i$).

expand_log

To apply identities 1 and 2 from left to right, use `expand_log()`. As always, the identities will not be applied unless they are valid.

```
>>> expand_log(log(x*y))
log(x) + log(y)
>>> expand_log(log(x/y))
log(x) - log(y)
>>> expand_log(log(x**2))
2*log(x)
>>> expand_log(log(x**n))
n*log(x)
>>> expand_log(log(z*t))
log(t*z)
```

As with `powsimp()` and `powdenest()`, `expand_log()` has a `force` option that can be used to ignore assumptions.

```
>>> expand_log(log(z**2))
 $\log(z^2)$ 
>>> expand_log(log(z**2), force=True)
2*log(z)
```

logcombine

To apply identities 1 and 2 from right to left, use `logcombine()`.

```
>>> logcombine(log(x) + log(y))
log(x*y)
>>> logcombine(n*log(x))
log(xn)
>>> logcombine(n*log(z))
n*log(z)
```

`logcombine()` also has a `force` option that can be used to ignore assumptions.

```
>>> logcombine(n*log(z), force=True)
log(zn)
```

2.6.6 Special Functions

SymPy implements dozens of special functions, ranging from functions in combinatorics to mathematical physics.

An extensive list of the special functions included with SymPy and their documentation is at the [Functions Module](#) (page 427) page.

For the purposes of this tutorial, let's introduce a few special functions in SymPy.

Let's define `x`, `y`, and `z` as regular, complex Symbols, removing any assumptions we put on them in the previous section. We will also define `k`, `m`, and `n`.

```
>>> x, y, z = symbols('x y z')
>>> k, m, n = symbols('k m n')
```

The `factorial` function is `factorial`. `factorial(n)` represents $n! = 1 \cdot 2 \cdots (n-1) \cdot n$. $n!$ represents the number of permutations of n distinct items.

```
>>> factorial(n)
n!
```

The `binomial coefficient` function is `binomial`. `binomial(n, k)` represents $\binom{n}{k}$, the number of ways to choose k items from a set of n distinct items. It is also often written as nCk , and is pronounced “ n choose k ”.

```
>>> binomial(n, k)

$$\binom{n}{k}$$

```

The `factorial` function is closely related to the `gamma function`, `gamma`. `gamma(z)` represents $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$, which for positive integer z is the same as $(z-1)!$.

```
>>> gamma(z)

$$\Gamma(z)$$

```

The [generalized hypergeometric function](#) is `hyper([a_1, ..., a_p], [b_1, ..., b_q], z)` represents ${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right)$. The most common case is ${}_2F_1$, which is often referred to as the [ordinary hypergeometric function](#).

```
>>> hyper([1, 2], [3], z)
```

$${}_2F_1 \left(\begin{matrix} 1, 2 \\ 3 \end{matrix} \middle| z \right)$$

rewrite

A common way to deal with special functions is to rewrite them in terms of one another. This works for any function in SymPy, not just special functions. To rewrite an expression in terms of a function, use `expr.rewrite(function)`. For example,

```
>>> tan(x).rewrite(sin)
```

$$\frac{\sin^2(x)}{\sin(x)}$$

$$\sin(2x)$$

```
>>> factorial(x).rewrite(gamma)
```

$$\Gamma(x + 1)$$

For some tips on applying more targeted rewriting, see the [Advanced Expression Manipulation](#) (page 55) section.

expand_func

To expand special functions in terms of some identities, use `expand_func()`. For example

```
>>> expand_func(gamma(x + 3))
```

$$x \cdot (x + 1) \cdot (x + 2) \cdot \Gamma(x)$$

hyperexpand

To rewrite `hyper` in terms of more standard functions, use `hyperexpand()`.

```
>>> hyperexpand(hyper([1, 1], [2], z))
```

$$-\log(-z + 1)$$

$$z$$

`hyperexpand()` also works on the more general Meijer G-function (see [its documentation](#) (page 532) for more information).

```
>>> expr = meijerg([[1],[1]], [[1],[1]], -z)
```

```
>>> expr
```

$$G_{2,1}^{1,1} \left(\begin{matrix} 1 & 1 \\ 1 \end{matrix} \middle| -z \right)$$

```
>>> hyperexpand(expr)
```

(continues on next page)

(continued from previous page)

```
1
-
z
e
```

combsimp

To simplify combinatorial expressions, use `combsimp()`.

```
>>> n, k = symbols('n k', integer = True)
>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
n + 1
-----
k + 1
```

gammasimp

To simplify expressions with gamma functions or combinatorial functions with non-integer argument, use `gammasimp()`.

```
>>> gammasimp(gamma(x)*gamma(1 - x))
      pi
-----
sin(pi*x)
```

2.6.7 Example: Continued Fractions

Let's use SymPy to explore continued fractions. A [continued fraction](#) is an expression of the form

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\ddots + \frac{1}{a_n}}}}$$

where $a_0, \dots, a_n$ are integers, and $a_1, \dots, a_n$ are positive. A continued fraction can also be infinite, but infinite objects are more difficult to represent in computers, so we will only examine the finite case here.

A continued fraction of the above form is often represented as a list $[a_0; a_1, \dots, a_n]$. Let's write a simple function that converts such a list to its continued fraction form. The easiest way to construct a continued fraction from a list is to work backwards. Note that despite the apparent symmetry of the definition, the first element, a_0 , must usually be handled differently from the rest.


```
>>> def list_to_frac(l):
...     expr = Integer(0)
...     for i in reversed(l[1:]):
...         expr += i
...         expr = 1/expr
...     return l[0] + expr
>>> list_to_frac([x, y, z])
1
x + ----
      1
      y + -
          z
```

We use `Integer(0)` in `list_to_frac` so that the result will always be a SymPy object, even if we only pass in Python ints.

```
>>> list_to_frac([1, 2, 3, 4])
43
—
30
```

Every finite continued fraction is a rational number, but we are interested in symbolics here, so let's create a symbolic continued fraction. The `symbols()` function that we have been using has a shortcut to create numbered symbols. `symbols('a0:5')` will create the symbols `a0`, `a1`, ..., `a4`.

```
>>> syms = symbols('a0:5')
>>> syms
(a0, a1, a2, a3, a4)
>>> a0, a1, a2, a3, a4 = syms
>>> frac = list_to_frac(syms)
>>> frac
1
a0 + ----
      1
      a1 + ----
            1
            a2 + ----
                  1
                  a3 + ----
                        1
                        a4
```

This form is useful for understanding continued fractions, but let's put it into standard rational function form using `cancel()`.

```
>>> frac = cancel(frac)
>>> frac
a0*a1*a2*a3*a4 + a0*a1*a2 + a0*a1*a4 + a0*a3*a4 + a0 + a2*a3*a4 + a2 + a4
-----
a1*a2*a3*a4 + a1*a2 + a1*a4 + a3*a4 + 1
```

Now suppose we were given `frac` in the above canceled form. In fact, we might be given the fraction in any form, but we can always put it into the above canonical form with `cancel()`. Suppose that we knew that it could be rewritten as a continued fraction. How could we do this with SymPy? A continued fraction is recursively $c + \frac{1}{f}$, where c is an integer and f is a (smaller) continued fraction. If we could write the expression in this form, we could pull out each c recursively and add it to a list. We could then get a continued fraction with our

`list_to_frac()` function.

The key observation here is that we can convert an expression to the form $c + \frac{1}{f}$ by doing a partial fraction decomposition with respect to c . This is because f does not contain c . This means we need to use the `apart()` function. We use `apart()` to pull the term out, then subtract it from the expression, and take the reciprocal to get the f part.

```
>>> l = []
>>> frac = apart(frac, a0)
>>> frac
a0 + 
$$\frac{a_2 \cdot a_3 \cdot a_4 + a_2 + a_4}{a_1 \cdot a_2 \cdot a_3 \cdot a_4 + a_1 \cdot a_2 + a_1 \cdot a_4 + a_3 \cdot a_4 + 1}$$

>>> l.append(a0)
>>> frac = 1/(frac - a0)
>>> frac

$$\frac{a_1 \cdot a_2 \cdot a_3 \cdot a_4 + a_1 \cdot a_2 + a_1 \cdot a_4 + a_3 \cdot a_4 + 1}{a_2 \cdot a_3 \cdot a_4 + a_2 + a_4}$$

```

Now we repeat this process

```
>>> frac = apart(frac, a1)
>>> frac
a1 + 
$$\frac{a_3 \cdot a_4 + 1}{a_2 \cdot a_3 \cdot a_4 + a_2 + a_4}$$

>>> l.append(a1)
>>> frac = 1/(frac - a1)
>>> frac = apart(frac, a2)
>>> frac
a2 + 
$$\frac{a_4}{a_3 \cdot a_4 + 1}$$

>>> l.append(a2)
>>> frac = 1/(frac - a2)
>>> frac = apart(frac, a3)
>>> frac
a3 + 
$$\frac{1}{a_4}$$

>>> l.append(a3)
>>> frac = 1/(frac - a3)
>>> frac = apart(frac, a4)
>>> frac
a4
>>> l.append(a4)
>>> list_to_frac(l)
a0 + 
$$\frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4}}}}$$

```

Quick Tip

You can execute multiple lines at once in SymPy Live. Typing Shift-Enter instead of Enter will enter a newline instead of executing.

Of course, this exercise seems pointless, because we already know that our `frac` is `list_to_frac([a0, a1, a2, a3, a4])`. So try the following exercise. Take a list of symbols and randomize them, and create the canceled continued fraction, and see if you can reproduce the original list. For example

```
>>> import random
>>> l = list(symbols('a0:5'))
>>> random.shuffle(l)
>>> orig_frac = frac = cancel(list_to_frac(l))
>>> del l
```

Click on “Run code block in SymPy Live” on the definition of `list_to_frac()` above, and then on the above example, and try to reproduce `l` from `frac`. I have deleted `l` at the end to remove the temptation for peeking (you can check your answer at the end by calling `cancel(list_to_frac(l))` on the list that you generate at the end, and comparing it to `orig_frac`).

See if you can think of a way to figure out what symbol to pass to `apart()` at each stage (hint: think of what happens to a_0 in the formula $a_0 + \frac{1}{a_1 + \dots}$ when it is canceled).

2.7 Calculus

This section covers how to do basic calculus tasks such as derivatives, integrals, limits, and series expansions in SymPy. If you are not familiar with the math of any part of this section, you may safely skip it.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

2.7.1 Derivatives

To take derivatives, use the `diff` function.

```
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
      ( 2 )
      (x )
2·x·e
```

`diff` can take multiple derivatives at once. To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example, both of the following find the third derivative of x^4 .

```
>>> diff(x**4, x, x, x)
24·x
>>> diff(x**4, x, 3)
24·x
```

You can also take derivatives with respect to many variables at once. Just pass each derivative in order, using the same syntax as for single variable derivatives. For example, each of the following will compute $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$.

```
>>> expr = exp(x*y*z)
>>> diff(expr, x, y, y, z, z, z, z)
3 2 ( 3 3 3      2 2 2 ) x·y·z
x ·y ·(x ·y ·z + 14·x ·y ·z + 52·x·y·z + 48)·e
>>> diff(expr, x, y, 2, z, 4)
3 2 ( 3 3 3      2 2 2 ) x·y·z
x ·y ·(x ·y ·z + 14·x ·y ·z + 52·x·y·z + 48)·e
>>> diff(expr, x, y, y, z, 4)
3 2 ( 3 3 3      2 2 2 ) x·y·z
x ·y ·(x ·y ·z + 14·x ·y ·z + 52·x·y·z + 48)·e
```

`diff` can also be called as a method. The two ways of calling `diff` are exactly the same, and are provided only for convenience.

```
>>> expr.diff(x, y, y, z, 4)
3 2 ( 3 3 3      2 2 2 ) x·y·z
x ·y ·(x ·y ·z + 14·x ·y ·z + 52·x·y·z + 48)·e
```

To create an unevaluated derivative, use the `Derivative` class. It has the same syntax as `diff`.

```
>>> deriv = Derivative(expr, x, y, y, z, 4)
>>> deriv
7
∂
----- ( x·y·z )
4 2
∂z ∂y ∂x e
```

To evaluate an unevaluated derivative, use the `doit` method.

```
>>> deriv.doit()
3 2 ( 3 3 3      2 2 2 ) x·y·z
x ·y ·(x ·y ·z + 14·x ·y ·z + 52·x·y·z + 48)·e
```

These unevaluated objects are useful for delaying the evaluation of the derivative, or for printing purposes. They are also used when SymPy does not know how to compute the derivative of an expression (for example, if it contains an undefined function, which are described in the [Solving Differential Equations](#) (page 47) section).

Derivatives of unspecified order can be created using tuple (x, n) where n is the order of the derivative with respect to x .

```
>>> m, n, a, b = symbols('m n a b')
>>> expr = (a*x + b)**m
>>> expr.diff((x, n))
n
```

(continues on next page)

(continued from previous page)

$$\frac{\partial}{\partial x} \left((a \cdot x + b)^m \right)$$

2.7.2 Integrals

To compute an integral, use the `integrate` function. There are two kinds of integrals, definite and indefinite. To compute an indefinite integral, that is, an antiderivative, or primitive, just pass the variable after the expression.

```
>>> integrate(cos(x), x)
sin(x)
```

Note that SymPy does not include the constant of integration. If you want it, you can add one yourself, or rephrase your problem as a differential equation and use `dsolve` to solve it, which does add the constant (see [Solving Differential Equations](#) (page 47)).

Quick Tip

∞ in SymPy is `oo` (that's the lowercase letter "oh" twice). This is because `oo` looks like ∞ , and is easy to type.

To compute a definite integral, pass the argument (`integration_variable`, `lower_limit`, `upper_limit`). For example, to compute

$$\int_0^{\infty} e^{-x} dx,$$

we would do

```
>>> integrate(exp(-x), (x, 0, oo))
1
```

As with indefinite integrals, you can pass multiple limit tuples to perform a multiple integral. For example, to compute

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} dx dy,$$

do

```
>>> integrate(exp(-x**2 - y**2), (x, -oo, oo), (y, -oo, oo))
pi
```

If `integrate` is unable to compute an integral, it returns an unevaluated `Integral` object.

```
>>> expr = integrate(x**x, x)
>>> print(expr)
Integral(x**x, x)
>>> expr
|
| x
```

(continues on next page)

(continued from previous page)

```
| x dx
|
```

As with `Derivative`, you can create an unevaluated integral using `Integral`. To later evaluate this integral, call `doit`.

```
>>> expr = Integral(log(x)**2, x)
>>> expr
|
|      2
| log (x) dx
|
>>> expr.doit()
      2
x·log (x) - 2·x·log(x) + 2·x
```

`integrate` uses powerful algorithms that are always improving to compute both definite and indefinite integrals, including heuristic pattern matching type algorithms, a partial implementation of the [Risch algorithm](#), and an algorithm using [Meijer G-functions](#) that is useful for computing integrals in terms of special functions, especially definite integrals. Here is a sampling of some of the power of `integrate`.

```
>>> integ = Integral((x**4 + x**2*exp(x) - x**2 - 2*x*exp(x) - 2*x -
... exp(x))*exp(x)/((x - 1)**2*(x + 1)**2*(exp(x) + 1)), x)
>>> integ
|
| ( 4      2 x      2      x      x ) x
| ( x  + x ·e  - x  - 2·x·e  - 2·x - e ) ·e
| ----- dx
|      2      2 ( x )
| (x - 1) · (x + 1) · (e  + 1)
|
>>> integ.doit()
      x
      e
log( e  + 1 ) + -----
      2
      x  - 1
```

```
>>> integ = Integral(sin(x**2), x)
>>> integ
|
| ( 2 )
| sin(x ) dx
|
>>> integ.doit()
      ( 2·x )
      √π
3·√2·√π·fresnels(-----)·Γ(3/4)
      √π
-----
      8·Γ(7/4)
```

```
>>> integ = Integral(x**y*exp(-x), (x, 0, oo))
>>> integ
```

(continues on next page)

(continued from previous page)

```


$$\int_0^{\infty} x^y \cdot e^{-x} dx$$

>>> integ.doit()

$$\begin{cases} \Gamma(y + 1) & \text{for } -\operatorname{re}(y) < 1 \\ \int_0^{\infty} x^y \cdot e^{-x} dx & \text{otherwise} \end{cases}$$


```

This last example returned a Piecewise expression because the integral does not converge unless $\Re(y) > -1$.

2.7.3 Limits

SymPy can compute symbolic limits with the `limit` function. The syntax to compute

$$\lim_{x \rightarrow x_0} f(x)$$

is `limit(f(x), x, x0)`.

```

>>> limit(sin(x)/x, x, 0)
1

```

`limit` should be used instead of `subs` whenever the point of evaluation is a singularity. Even though SymPy has objects to represent ∞ , using them for evaluation is not reliable because they do not keep track of things like rate of growth. Also, things like $\infty - \infty$ and $\frac{\infty}{\infty}$ return `nan` (not-a-number). For example

```

>>> expr = x**2/exp(x)
>>> expr.subs(x, oo)
nan
>>> limit(expr, x, oo)
0

```

Like `Derivative` and `Integral`, `limit` has an unevaluated counterpart, `Limit`. To evaluate it, use `doit`.

```

>>> expr = Limit((cos(x) - 1)/x, x, 0)
>>> expr

$$\lim_{x \rightarrow 0^+} \left| \frac{\cos(x) - 1}{x} \right|$$

>>> expr.doit()
0

```

To evaluate a limit at one side only, pass '+' or '-' as a third argument to `limit`. For example, to compute

$$\lim_{x \rightarrow 0^+} \frac{1}{x},$$

do

```
>>> limit(1/x, x, 0, '+')
∞
```

As opposed to

```
>>> limit(1/x, x, 0, '-')
-∞
```

2.7.4 Series Expansion

SymPy can compute asymptotic series expansions of functions around a point. To compute the expansion of $f(x)$ around the point $x = x_0$ terms of order x^n , use `f(x).series(x, x0, n)`. `x0` and `n` can be omitted, in which case the defaults `x0=0` and `n=6` will be used.

```
>>> expr = exp(sin(x))
>>> expr.series(x, 0, 4)
      2
      x
1 + x + — + O(x4)
      2
```

The $O(x^4)$ term at the end represents the Landau order term at $x = 0$ (not to be confused with big O notation used in computer science, which generally represents the Landau order term at $x = \infty$). It means that all x terms with power greater than or equal to x^4 are omitted. Order terms can be created and manipulated outside of `series`. They automatically absorb higher order terms.

```
>>> x + x**3 + x**6 + O(x**4)
      3      ( 4)
x + x  + O(x )
>>> x*O(1)
O(x)
```

If you do not want the order term, use the `removeO` method.

```
>>> expr.series(x, 0, 4).removeO()
      2
      x
— + x + 1
      2
```

The `O` notation supports arbitrary limit points (other than 0):

```
>>> exp(x - 6).series(x, x0=6)
      2      3      4      5
      (x - 6)  (x - 6)  (x - 6)  (x - 6)
-5 + — + — + — + — + x + O((x - 6)6; x → 6)
      2      6      24     120
```


2.7.5 Finite differences

So far we have looked at expressions with analytic derivatives and primitive functions respectively. But what if we want to have an expression to estimate a derivative of a curve for which we lack a closed form representation, or for which we don't know the functional values for yet. One approach would be to use a finite difference approach.

The simplest way to differentiate using finite differences is to use the `differentiate_finite` function:

```
>>> f, g = symbols('f g', cls=Function)
>>> differentiate_finite(f(x)*g(x))
-f(x - 1/2)*g(x - 1/2) + f(x + 1/2)*g(x + 1/2)
```

If we want to expand the intermediate derivative we may pass the flag `evaluate=True`:

```
>>> differentiate_finite(f(x)*g(x), evaluate=True)
(-f(x - 1/2) + f(x + 1/2))*g(x) + (-g(x - 1/2) + g(x + 1/2))*f(x)
```

This form however does not respect the product rule.

If you already have a `Derivative` instance, you can use the `as_finite_difference` method to generate approximations of the derivative to arbitrary order:

```
>>> f = Function('f')
>>> dfdx = f(x).diff(x)
>>> dfdx.as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

here the first order derivative was approximated around x using a minimum number of points (2 for 1st order derivative) evaluated equidistantly using a step-size of 1. We can use arbitrary steps (possibly containing symbolic expressions):

```
>>> f = Function('f')
>>> d2fdx2 = f(x).diff(x, 2)
>>> h = Symbol('h')
>>> d2fdx2.as_finite_difference([-3*h, -h, 2*h])
f(-3*h)  f(-h)  2*f(2*h)
----- - ----- + -----
      2      2      2
    5*h    3*h   15*h
```

If you are just interested in evaluating the weights, you can do so manually:

```
>>> finite_diff_weights(2, [-3, -1, 2], 0)[-1][-1]
[1/5, -1/3, 2/15]
```

note that we only need the last element in the last sublist returned from `finite_diff_weights`. The reason for this is that the function also generates weights for lower derivatives and using fewer points (see the documentation of `finite_diff_weights` for more details).

If using `finite_diff_weights` directly looks complicated, and the `as_finite_difference` method of `Derivative` instances is not flexible enough, you can use `apply_finite_diff` which takes `order`, `x_list`, `y_list` and `x0` as parameters:

```
>>> x_list = [-3, 1, 2]
>>> y_list = symbols('a b c')
```

(continues on next page)

(continued from previous page)

```
>>> apply_finite_diff(1, x_list, y_list, 0)
      3·a    b    2·c
-  ----- -  -  +  -----
      20    4    5
```

2.8 Solvers

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

2.8.1 A Note about Equations

Recall from the [gotchas](#) (page 10) section of this tutorial that symbolic equations in SymPy are not represented by `=` or `==`, but by `Eq`.

```
>>> Eq(x, y)
x = y
```

However, there is an even easier way. In SymPy, any expression not in an `Eq` is automatically assumed to equal 0 by the solving functions. Since $a = b$ if and only if $a - b = 0$, this means that instead of using `x == y`, you can just use `x - y`. For example

```
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
>>> solveset(Eq(x**2 - 1, 0), x)
{-1, 1}
>>> solveset(x**2 - 1, x)
{-1, 1}
```

This is particularly useful if the equation you wish to solve is already equal to 0. Instead of typing `solveset(Eq(expr, 0), x)`, you can just use `solveset(expr, x)`.

2.8.2 Solving Equations Algebraically

The main function for solving algebraic equations is `solveset`. The syntax for `solveset` is `solveset(equation, variable=None, domain=S.Complexes)` Where equations may be in the form of `Eq` instances or expressions that are assumed to be equal to zero.

Please note that there is another function called `solve` which can also be used to solve equations. The syntax is `solve(equations, variables)` However, it is recommended to use `solveset` instead.

When solving a single equation, the output of `solveset` is a `FiniteSet` or an `Interval` or `ImageSet` of the solutions.

```
>>> solveset(x**2 - x, x)
{0, 1}
>>> solveset(x - x, x, domain=S.Reals)
ℝ
```

(continues on next page)

(continued from previous page)

```
>>> solveset(sin(x) - 1, x, domain=S.Reals)
{2·n·π +  $\frac{\pi}{2}$  | n ∈ ℤ}
```

If there are no solutions, an `EmptySet` is returned and if it is not able to find solutions then a `ConditionSet` is returned.

```
>>> solveset(exp(x), x)      # No solution exists
∅
>>> solveset(cos(x) - x, x)  # Not able to find solution
{x | x ∈ ℂ ∧ -x + cos(x) = 0}
```

In the `solveset` module, the linear system of equations is solved using `linsolve`. In future we would be able to use `linsolve` directly from `solveset`. Following is an example of the syntax of `linsolve`.

- List of Equations Form:

```
>>> linsolve([x + y + z - 1, x + y + 2*z - 3], (x, y, z))
{(-y - 1, y, 2)}
```

- Augmented Matrix Form:

```
>>> linsolve(Matrix([[1, 1, 1, 1], [1, 1, 2, 3]]), (x, y, z))
{(-y - 1, y, 2)}
```

- $A \cdot x = b$ Form

```
>>> M = Matrix([(1, 1, 1, 1), (1, 1, 2, 3)])
>>> system = A, b = M[:, :-1], M[:, -1]
>>> linsolve(system, x, y, z)
{(-y - 1, y, 2)}
```

Note: The order of solution corresponds the order of given symbols.

In the `solveset` module, the non linear system of equations is solved using `nonlinsolve`. Following are examples of `nonlinsolve`.

1. When only real solution is present:

```
>>> a, b, c, d = symbols('a, b, c, d', real=True)
>>> nonlinsolve([a**2 + a, a - b], [a, b])
{(-1, -1), (0, 0)}
>>> nonlinsolve([x*y - 1, x - 2], x, y)
{(2, 1/2)}
```

2. When only complex solution is present:

```
>>> nonlinsolve([x**2 + 1, y**2 + 1], [x, y])
{(-i, -i), (-i, i), (i, -i), (i, i)}
```

3. When both real and complex solution is present:

```
>>> from sympy import sqrt
>>> system = [x**2 - 2*y**2 - 2, x*y - 2]
>>> vars = [x, y]
>>> nonlinsolve(system, vars)
{(-2, -1), (2, 1), (-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I)}
```

```
>>> n = Dummy('n')
>>> system = [exp(x) - sin(y), 1/y - 3]
>>> real_soln = (log(sin(S(1)/3)), S(1)/3)
>>> img_lambda = Lambda(n, 2*n*I*pi + Mod(log(sin(S(1)/3)), 2*I*pi))
>>> complex_soln = (ImageSet(img_lambda, S.Integers), S(1)/3)
>>> soln = FiniteSet(real_soln, complex_soln)
>>> nonlinsolve(system, [x, y]) == soln
True
```

4. If non linear system of equations is Positive dimensional system (A system with infinitely many solutions is said to be positive-dimensional):

```
>>> nonlinsolve([x*y, x*y - x], [x, y])
{(0, y)}
```

```
>>> system = [a**2 + a*c, a - b]
>>> nonlinsolve(system, [a, b])
{(0, 0), (-c, -c)}
```

Note:

1. The order of solution corresponds the order of given symbols.
2. Currently nonlinsolve doesn't return solution in form of LambertW (if there is solution present in the form of LambertW).

solve can be used for such cases:

```
>>> solve([x**2 - y**2/exp(x)], [x, y], dict=True)
[[{x: 2*LambertW(y/2)}]]
```

3. Currently nonlinsolve is not properly capable of solving the system of equations having trigonometric functions.

solve can be used for such cases(not all solution):

```
>>> solve([sin(x + y), cos(x - y)], [x, y])
[[(-3*pi/4, 3*pi/4), (-pi/4, pi/4), (pi/4, 3*pi/4), (3*pi/4, pi/4)]]
```

solveset reports each solution only once. To get the solutions of a polynomial including multiplicity use roots.

```
>>> solveset(x**3 - 6*x**2 + 9*x, x)
{0, 3}
>>> roots(x**3 - 6*x**2 + 9*x, x)
{0: 1, 3: 2}
```

The output `{0: 1, 3: 2}` of roots means that 0 is a root of multiplicity 1 and 3 is a root of multiplicity 2.

Note: Currently `solveset` is not capable of solving the following types of equations:

- Equations solvable by `LambertW` (Transcendental equation solver).

`solve` can be used for such cases:

```
>>> solve(x*exp(x) - 1, x)
[LambertW(1)]
```

2.8.3 Solving Differential Equations

To solve differential equations, use `dsolve`. First, create an undefined function by passing `cls=Function` to the `symbols` function.

```
>>> f, g = symbols('f g', cls=Function)
```

`f` and `g` are now undefined functions. We can call `f(x)`, and it will represent an unknown function.

```
>>> f(x)
f(x)
```

Derivatives of `f(x)` are unevaluated.

```
>>> f(x).diff(x)
d
—(f(x))
dx
```

(see the [Derivatives](#) (page 37) section for more on derivatives).

To represent the differential equation $f''(x) - 2f'(x) + f(x) = \sin(x)$, we would thus use

```
>>> diffeq = Eq(f(x).diff(x, x) - 2*f(x).diff(x) + f(x), sin(x))
>>> diffeq
```

$$f(x) - 2 \frac{d}{dx}(f(x)) + \frac{d^2}{dx^2}(f(x)) = \sin(x)$$

To solve the ODE, pass it and the function to solve for to `dsolve`.

```
>>> dsolve(diffeq, f(x))
f(x) = (C1 + C2*x)*ex +  $\frac{\cos(x)}{2}$ 
```

`dsolve` returns an instance of `Eq`. This is because in general, solutions to differential equations cannot be solved explicitly for the function.

```
>>> dsolve(f(x).diff(x)*(1 - sin(f(x))), f(x))
f(x) + cos(f(x)) = C1
```

The arbitrary constants in the solutions from `dsolve` are symbols of the form `C1`, `C2`, `C3`, and so on.

2.9 Matrices

```
>>> from sympy import *
>>> init_printing(use_unicode=True)
```

To make a matrix in SymPy, use the `Matrix` object. A matrix is constructed by providing a list of row vectors that make up the matrix. For example, to construct the matrix

$$\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 0 & 2 \end{bmatrix}$$

use

```
>>> Matrix([[1, -1], [3, 4], [0, 2]])
⎡ 1  -1 ⎤
⎢ 3   4 ⎢
⎣ 0   2 ⎣
```

To make it easy to make column vectors, a list of elements is considered to be a column vector.

```
>>> Matrix([1, 2, 3])
⎡ 1 ⎤
⎢ 2 ⎤
⎣ 3 ⎤
```

Matrices are manipulated just like any other object in SymPy or Python.

```
>>> M = Matrix([[1, 2, 3], [3, 2, 1]])
>>> N = Matrix([0, 1, 1])
>>> M*N
⎡ 5 ⎤
⎢  ⎤
⎣ 3 ⎤
```

One important thing to note about SymPy matrices is that, unlike every other object in SymPy, they are mutable. This means that they can be modified in place, as we will see below. The downside to this is that `Matrix` cannot be used in places that require immutability, such as inside other SymPy expressions or as keys to dictionaries. If you need an immutable version of `Matrix`, use `ImmutableMatrix`.

2.9.1 Basic Operations

Shape

Here are some basic operations on `Matrix`. To get the shape of a matrix use `shape`

```
>>> M = Matrix([[1, 2, 3], [-2, 0, 4]])
>>> M
[1  2  3]
[-2 0  4]
>>> M.shape
(2, 3)
```

Accessing Rows and Columns

To get an individual row or column of a matrix, use `row` or `col`. For example, `M.row(0)` will get the first row. `M.col(-1)` will get the last column.

```
>>> M.row(0)
[1  2  3]
>>> M.col(-1)
[3]
[4]
```

Deleting and Inserting Rows and Columns

To delete a row or column, use `row_del` or `col_del`. These operations will modify the Matrix **in place**.

```
>>> M.col_del(0)
>>> M
[2  3]
[0  4]
>>> M.row_del(1)
>>> M
[2  3]
```

To insert rows or columns, use `row_insert` or `col_insert`. These operations **do not** operate in place.

```
>>> M
[2  3]
>>> M = M.row_insert(1, Matrix([[0, 4]]))
>>> M
[2  3]
[0  4]
>>> M = M.col_insert(0, Matrix([1, -2]))
>>> M
[1  2  3]
[-2 0  4]
```

Unless explicitly stated, the methods mentioned below do not operate in place. In general, a method that does not operate in place will return a new `Matrix` and a method that does operate in place will return `None`.

2.9.2 Basic Methods

As noted above, simple operations like addition and multiplication are done just by using `+`, `*`, and `**`. To find the inverse of a matrix, just raise it to the `-1` power.

```
>>> M = Matrix([[1, 3], [-2, 3]])
>>> N = Matrix([[0, 3], [0, 7]])
>>> M + N
[1  6]
[-2 10]
>>> M*N
[0 24]
[0 15]
>>> 3*M
[3  9]
[-6 9]
>>> M**2
[-5 12]
[-8  3]
>>> M**-1
[1/3 -1/3]
[2/9  1/9]
>>> N**-1
Traceback (most recent call last):
...
ValueError: Matrix det == 0; not invertible.
```

To take the transpose of a Matrix, use `T`.

```
>>> M = Matrix([[1, 2, 3], [4, 5, 6]])
>>> M
[1 2 3]
[4 5 6]
>>> M.T
[1 4]
[2 5]
[3 6]
```

2.9.3 Matrix Constructors

Several constructors exist for creating common matrices. To create an identity matrix, use `eye`. `eye(n)` will create an $n \times n$ identity matrix.

```
>>> eye(3)
[1 0 0]
|   |
```

(continues on next page)

(continued from previous page)

```

[0 1 0]
[0 0 1]
>>> eye(4)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

```

To create a matrix of all zeros, use `zeros`. `zeros(n, m)` creates an $n \times m$ matrix of 0s.

```

>>> zeros(2, 3)
[0 0 0]
[0 0 0]

```

Similarly, `ones` creates a matrix of ones.

```

>>> ones(3, 2)
[1 1]
[1 1]
[1 1]

```

To create diagonal matrices, use `diag`. The arguments to `diag` can be either numbers or matrices. A number is interpreted as a 1×1 matrix. The matrices are stacked diagonally. The remaining elements are filled with 0s.

```

>>> diag(1, 2, 3)
[1 0 0]
[0 2 0]
[0 0 3]
>>> diag(-1, ones(2, 2), Matrix([5, 7, 5]))
[-1 0 0 0]
[0 1 1 0]
[0 1 1 0]
[0 0 0 5]
[0 0 0 7]
[0 0 0 5]

```

2.9.4 Advanced Methods

Determinant

To compute the determinant of a matrix, use `det`.

```
>>> M = Matrix([[1, 0, 1], [2, -1, 3], [4, 3, 2]])
>>> M

$$\begin{bmatrix} 1 & 0 & 1 \\ 2 & -1 & 3 \\ 4 & 3 & 2 \end{bmatrix}$$

>>> M.det()
-1
```

RREF

To put a matrix into reduced row echelon form, use `rref`. `rref` returns a tuple of two elements. The first is the reduced row echelon form, and the second is a tuple of indices of the pivot columns.

```
>>> M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])
>>> M

$$\begin{bmatrix} 1 & 0 & 1 & 3 \\ 2 & 3 & 4 & 7 \\ -1 & -3 & -3 & -4 \end{bmatrix}$$

>>> M.rref()

$$\left( \begin{bmatrix} 1 & 0 & 1 & 3 \\ 0 & 1 & 2/3 & 1/3 \\ 0 & 0 & 0 & 0 \end{bmatrix}, (0, 1) \right)$$

```

Note: The first element of the tuple returned by `rref` is of type `Matrix`. The second is of type `tuple`.

Nullspace

To find the nullspace of a matrix, use `nullspace`. `nullspace` returns a list of column vectors that span the nullspace of the matrix.

```
>>> M = Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])
>>> M

$$\begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 10 & 0 & 0 & 1 \end{bmatrix}$$

>>> M.nullspace()
```

(continues on next page)

(continued from previous page)

$$\begin{bmatrix} -15 \\ 6 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ -1/2 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Columnspace

To find the column space of a matrix, use `columnspace`. `columnspace` returns a list of column vectors that span the column space of the matrix.

```
>>> M = Matrix([[1, 1, 2], [2, 1, 3], [3, 1, 4]])
>>> M
[1 1 2]
[2 1 3]
[3 1 4]
>>> M.columnspace()
[[1] [1]]
[2] [1]
[3] [1]]
```

Eigenvalues, Eigenvectors, and Diagonalization

To find the eigenvalues of a matrix, use `eigenvals`. `eigenvals` returns a dictionary of eigenvalue:algebraic multiplicity pairs (similar to the output of `roots` (page 46)).

```
>>> M = Matrix([[3, -2, 4, -2], [5, 3, -3, -2], [5, -2, 2, -2], [5, -2, -3, 3]])
>>> M
[3 -2 4 -2]
[5 3 -3 -2]
[5 -2 2 -2]
[5 -2 -3 3]
>>> M.eigenvals()
{-2: 1, 3: 1, 5: 2}
```

This means that `M` has eigenvalues -2, 3, and 5, and that the eigenvalues -2 and 3 have algebraic multiplicity 1 and that the eigenvalue 5 has algebraic multiplicity 2.

To find the eigenvectors of a matrix, use `eigenvects`. `eigenvects` returns a list of tuples of the form (eigenvalue:algebraic multiplicity, [eigenvectors]).

```
>>> M.eigenvects()
([(-2, 1, [[0], [1], [1], [1]]), (3, 1, [[1], [1], [1], [1]]), (5, 2, [[1], [1], [1], [0]]), (5, 2, [[0], [-1], [0], [1]])])
```

This shows us that, for example, the eigenvalue 5 also has geometric multiplicity 2, because it has two eigenvectors. Because the algebraic and geometric multiplicities are the same for all the eigenvalues, M is diagonalizable.

To diagonalize a matrix, use `diagonalize`. `diagonalize` returns a tuple (P, D) , where D is diagonal and $M = PDP^{-1}$.

```
>>> P, D = M.diagonalize()
>>> P
[[0 1 1 0]
 [1 1 1 -1]
 [1 1 1 0]
 [1 1 0 1]]
>>> D
[[-2 0 0 0]
 [0 3 0 0]
 [0 0 5 0]
 [0 0 0 5]]
>>> P*D*P**-1
[[3 -2 4 -2]
 [5 3 -3 -2]
 [5 -2 2 -2]
 [5 -2 -3 3]]
>>> P*D*P**-1 == M
True
```

Quick Tip

`lambda` is a reserved keyword in Python, so to create a Symbol called λ , while using the same names for SymPy Symbols and Python variables, use `lamda` (without the b). It will still pretty print as λ .

Note that since `eigenvects` also includes the eigenvalues, you should use it instead of `eigenvals` if you also want the eigenvectors. However, as computing the eigenvectors may often be costly, `eigenvals` should be preferred if you only wish to find the eigenvalues.

If all you want is the characteristic polynomial, use `charpoly`. This is more efficient than `eigenvals`, because sometimes symbolic roots can be expensive to calculate.

```
>>> lamda = symbols('lamda')
>>> p = M.charpoly(lamda)
>>> factor(p)
      2
(\lambda - 5) \cdot (\lambda - 3) \cdot (\lambda + 2)
```

2.10 Advanced Expression Manipulation

In this section, we discuss some ways that we can perform advanced manipulation of expressions.

2.10.1 Understanding Expression Trees

Quick Tip

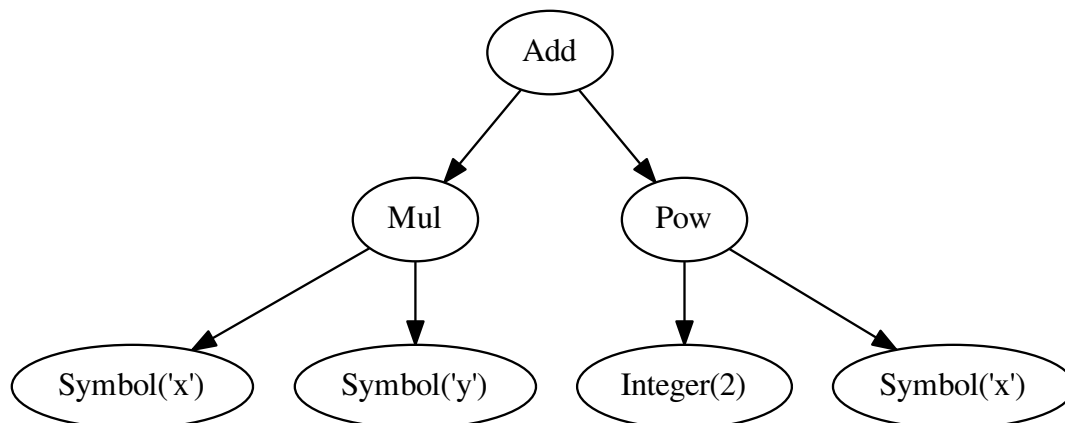
To play with the `srepr` form of expressions in the SymPy Live shell, change the output format to `Repr` in the settings.

Before we can do this, we need to understand how expressions are represented in SymPy. A mathematical expression is represented as a tree. Let us take the expression $2^x + xy$, i.e., `2**x + x*y`. We can see what this expression looks like internally by using `srepr`

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
```

```
>>> expr = 2**x + x*y
>>> srepr(expr)
"Add(Pow(Integer(2), Symbol('x')), Mul(Symbol('x'), Symbol('y')))"
```

The easiest way to tear this apart is to look at a diagram of the expression tree:



Note: The above diagram was made using [Graphviz](#) and the [dotprint](#) (page 1058) function.

First, let's look at the leaves of this tree. Symbols are instances of the class `Symbol`. While we have been doing

```
>>> x = symbols('x')
```

we could have also done

```
>>> x = Symbol('x')
```

Either way, we get a `Symbol` with the name “x”¹. For the number in the expression, 2, we got `Integer(2)`. `Integer` is the SymPy class for integers. It is similar to the Python built-in type `int`, except that `Integer` plays nicely with other SymPy types.

When we write `2**x`, this creates a `Pow` object. `Pow` is short for “power”.

```
>>> srepr(2**x)
"Pow(Integer(2), Symbol('x'))"
```

We could have created the same object by calling `Pow(2, x)`

```
>>> Pow(2, x)
2**x
```

Note that in the `srepr` output, we see `Integer(2)`, the SymPy version of integers, even though technically, we input 2, a Python `int`. In general, whenever you combine a SymPy object with a non-SymPy object via some function or operation, the non-SymPy object will be converted into a SymPy object. The function that does this is `sympify`².

¹ We have been using `symbols` instead of `Symbol` because it automatically splits apart strings into multiple `Symbols`. `symbols('x y z')` returns a tuple of three `Symbols`. `Symbol('x y z')` returns a single `Symbol` called `x y z`.

² Technically, it is an internal function called `_sympify`, which differs from `sympify` in that it does not convert strings. `x + '2'` is not allowed.

```
>>> type(2)
<... 'int'>
>>> type(sympify(2))
<class 'sympy.core.numbers.Integer'>
```

We have seen that $2**x$ is represented as `Pow(2, x)`. What about $x*y$? As we might expect, this is the multiplication of x and y . The SymPy class for multiplication is `Mul`.

```
>>> srepr(x*y)
"Mul(Symbol('x'), Symbol('y'))"
```

Thus, we could have created the same object by writing `Mul(x, y)`.

```
>>> Mul(x, y)
x*y
```

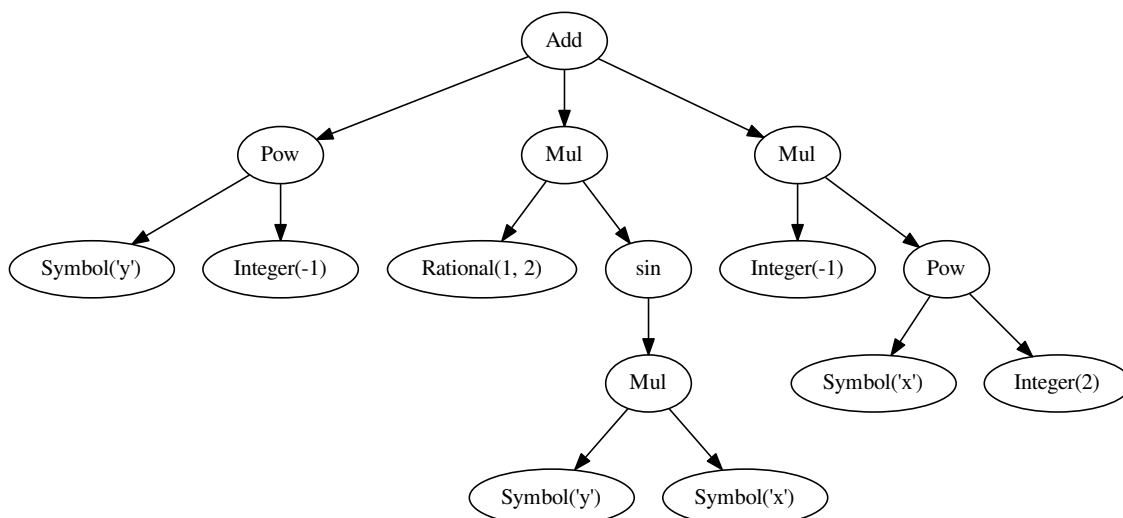
Now we get to our final expression, $2**x + x*y$. This is the addition of our last two objects, `Pow(2, x)`, and `Mul(x, y)`. The SymPy class for addition is `Add`, so, as you might expect, to create this object, we use `Add(Pow(2, x), Mul(x, y))`.

```
>>> Add(Pow(2, x), Mul(x, y))
2**x + x*y
```

SymPy expression trees can have many branches, and can be quite deep or quite broad. Here is a more complicated example

```
>>> expr = sin(x*y)/2 - x**2 + 1/y
>>> srepr(expr)
"Add(Mul(Integer(-1), Pow(Symbol('x'), Integer(2))), Mul(Rational(1, 2),
sin(Mul(Symbol('x'), Symbol('y')))), Pow(Symbol('y'), Integer(-1)))"
```

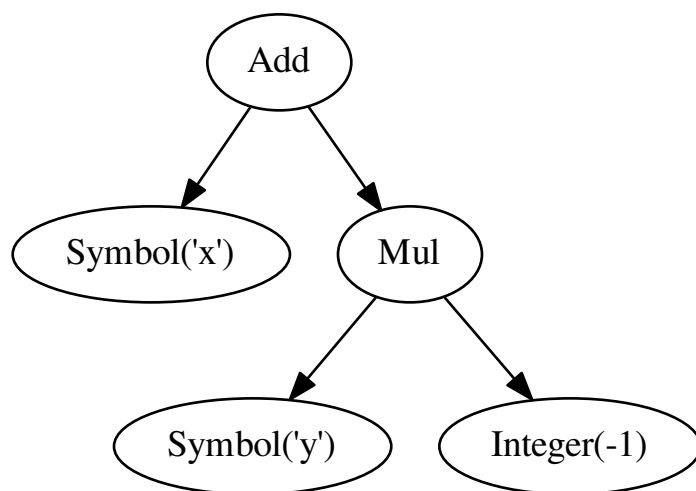
Here is a diagram



This expression reveals some interesting things about SymPy expression trees. Let's go through them one by one.

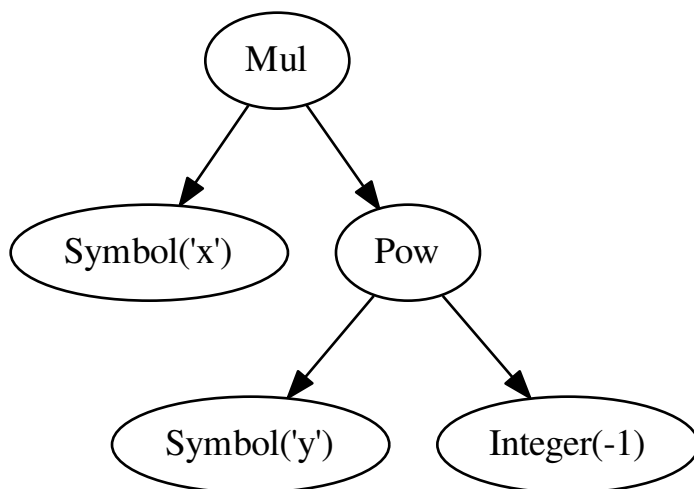
Let's first look at the term x^2 . As we expected, we see `Pow(x, 2)`. One level up, we see we have `Mul(-1, Pow(x, 2))`. There is no subtraction class in SymPy. $x - y$ is represented as $x + -y$, or, more completely, $x + -1*y$, i.e., `Add(x, Mul(-1, y))`.

```
>>> srepr(x - y)
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```



Next, look at $1/y$. We might expect to see something like `Div(1, y)`, but similar to subtraction, there is no class in SymPy for division. Rather, division is represented by a power of -1. Hence, we have `Pow(y, -1)`. What if we had divided something other than 1 by y , like x/y ? Let's see.

```
>>> expr = x/y
>>> srepr(expr)
"Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```

We see that x/y is represented as $x*y**-1$, i.e., `Mul(x, Pow(y, -1))`.

Finally, let's look at the $\sin(x*y)/2$ term. Following the pattern of the previous example, we might expect to see `Mul(sin(x*y), Pow(Integer(2), -1))`. But instead, we have `Mul(Rational(1, 2), sin(x*y))`. Rational numbers are always combined into a single term in a multiplication, so that when we divide by 2, it is represented as multiplying by $1/2$.

Finally, one last note. You may have noticed that the order we entered our expression and the order that it came out from `srepr` or in the graph were different. You may have also noticed this phenomenon earlier in the tutorial. For example

```
>>> 1 + x
x + 1
```

This because in SymPy, the arguments of the commutative operations `Add` and `Mul` are stored in an arbitrary (but consistent!) order, which is independent of the order inputted (if you're worried about noncommutative multiplication, don't be. In SymPy, you can create noncommutative Symbols using `Symbol('A', commutative=False)`, and the order of multiplication for noncommutative Symbols is kept the same as the input). Furthermore, as we shall see in the next section, the printing order and the order in which things are stored internally need not be the same either.

Quick Tip

The way an expression is represented internally and the way it is printed are often not the same.

In general, an important thing to keep in mind when working with SymPy expression trees is this: the internal representation of an expression and the way it is printed need not be the same. The same is true for the input form. If some expression manipulation algorithm is not working in the way you expected it to, chances are, the internal representation of the object is different from what you thought it was.

2.10.2 Recursing through an Expression Tree

Now that you know how expression trees work in SymPy, let's look at how to dig our way through an expression tree. Every object in SymPy has two very important attributes, `func`, and `args`.

`func`

`func` is the head of the object. For example, `(x*y).func` is `Mul`. Usually it is the same as the class of the object (though there are exceptions to this rule).

Two notes about `func`. First, the class of an object need not be the same as the one used to create it. For example

```
>>> expr = Add(x, x)
>>> expr.func
<class 'sympy.core.mul.Mul'>
```

We created `Add(x, x)`, so we might expect `expr.func` to be `Add`, but instead we got `Mul`. Why is that? Let's take a closer look at `expr`.

```
>>> expr
2*x
```

`Add(x, x)`, i.e., $x + x$, was automatically converted into `Mul(2, x)`, i.e., $2*x$, which is a `Mul`. SymPy classes make heavy use of the `__new__` class constructor, which, unlike `__init__`, allows a different class to be returned from the constructor.

Second, some classes are special-cased, usually for efficiency reasons³.

```
>>> Integer(2).func
<class 'sympy.core.numbers.Integer'>
>>> Integer(0).func
<class 'sympy.core.numbers.Zero'>
>>> Integer(-1).func
<class 'sympy.core.numbers.NegativeOne'>
```

For the most part, these issues will not bother us. The special classes `Zero`, `One`, `NegativeOne`, and so on are subclasses of `Integer`, so as long as you use `isinstance`, it will not be an issue.

`args`

`args` are the top-level arguments of the object. `(x*y).args` would be `(x, y)`. Let's look at some examples

```
>>> expr = 3*y**2*x
>>> expr.func
<class 'sympy.core.mul.Mul'>
>>> expr.args
(3, x, y**2)
```

³ Classes like `One` and `Zero` are singletonized, meaning that only one object is ever created, no matter how many times the class is called. This is done for space efficiency, as these classes are very common. For example, `Zero` might occur very often in a sparse matrix represented densely. As we have seen, `NegativeOne` occurs any time we have $-x$ or $1/x$. It is also done for speed efficiency because singletonized objects can be compared by `is`. The unique objects for each singletonized class can be accessed from the `S` object.

From this, we can see that `expr == Mul(3, y**2, x)`. In fact, we can see that we can completely reconstruct `expr` from its `func` and its `args`.

```
>>> expr.func(*expr.args)
3*x*y**2
>>> expr == expr.func(*expr.args)
True
```

Note that although we entered `3*y**2*x`, the `args` are `(3, x, y**2)`. In a `Mul`, the Rational coefficient will come first in the `args`, but other than that, the order of everything else follows no special pattern. To be sure, though, there is an order.

```
>>> expr = y**2*3*x
>>> expr.args
(3, x, y**2)
```

`Mul`'s `args` are sorted, so that the same `Mul` will have the same `args`. But the sorting is based on some criteria designed to make the sorting unique and efficient that has no mathematical significance.

The `srepr` form of our `expr` is `Mul(3, x, Pow(y, 2))`. What if we want to get at the `args` of `Pow(y, 2)`. Notice that the `y**2` is in the third slot of `expr.args`, i.e., `expr.args[2]`.

```
>>> expr.args[2]
y**2
```

So to get the `args` of this, we call `expr.args[2].args`.

```
>>> expr.args[2].args
(y, 2)
```

Now what if we try to go deeper. What are the `args` of `y`. Or `2`. Let's see.

```
>>> y.args
()
>>> Integer(2).args
()
```

They both have empty `args`. In SymPy, empty `args` signal that we have hit a leaf of the expression tree.

So there are two possibilities for a SymPy expression. Either it has empty `args`, in which case it is a leaf node in any expression tree, or it has `args`, in which case, it is a branch node of any expression tree. When it has `args`, it can be completely rebuilt from its `func` and its `args`. This is expressed in the key invariant.

Key Invariant

Every well-formed SymPy expression must either have empty `args` or satisfy `expr == expr.func(*expr.args)`.

(Recall that in Python if `a` is a tuple, then `f(*a)` means to call `f` with arguments from the elements of `a`, e.g., `f(*(1, 2, 3))` is the same as `f(1, 2, 3)`.)

This key invariant allows us to write simple algorithms that walk expression trees, change them, and rebuild them into new expressions.

Walking the Tree

With this knowledge, let's look at how we can recurse through an expression tree. The nested nature of args is a perfect fit for recursive functions. The base case will be empty args. Let's write a simple function that goes through an expression and prints all the args at each level.

```
>>> def pre(expr):  
...     print(expr)  
...     for arg in expr.args:  
...         pre(arg)
```

See how nice it is that `()` signals leaves in the expression tree. We don't even have to write a base case for our recursion; it is handled automatically by the for loop.

Let's test our function.

```
>>> expr = x*y + 1  
>>> pre(expr)  
x*y + 1  
1  
x*y  
x  
y
```

Can you guess why we called our function `pre`? We just wrote a pre-order traversal function for our expression tree. See if you can write a post-order traversal function.

Such traversals are so common in SymPy that the generator functions `preorder_traversal` and `postorder_traversal` are provided to make such traversals easy. We could have also written our algorithm as

```
>>> for arg in preorder_traversal(expr):  
...     print(arg)  
x*y + 1  
1  
x*y  
x  
y
```

2.10.3 Prevent expression evaluation

There are generally two ways to prevent the evaluation, either pass an `evaluate=False` parameter while constructing the expression, or create an evaluation stopper by wrapping the expression with `UnevaluatedExpr`.

For example:

```
>>> from sympy import Add  
>>> from sympy.abc import x, y, z  
>>> x + x  
2*x  
>>> Add(x, x)  
2*x  
>>> Add(x, x, evaluate=False)  
x + x
```

If you don't remember the class corresponding to the expression you want to build (operator overloading usually assumes `evaluate=True`), just use `sympify` and pass a string:

```
>>> from sympy import sympify
>>> sympify("x + x", evaluate=False)
x + x
```

Note that `evaluate=False` won't prevent future evaluation in later usages of the expression:

```
>>> expr = Add(x, x, evaluate=False)
>>> expr
x + x
>>> expr + x
3*x
```

That's why the class `UnevaluatedExpr` comes handy. `UnevaluatedExpr` is a method provided by SymPy which lets the user keep an expression unevaluated. By `unevaluated` it is meant that the value inside of it will not interact with the expressions outside of it to give simplified outputs. For example:

```
>>> from sympy import UnevaluatedExpr
>>> expr = x + UnevaluatedExpr(x)
>>> expr
x + x
>>> x + expr
2*x + x
```

The x remaining alone is the x wrapped by `UnevaluatedExpr`. To release it:

```
>>> (x + expr).doit()
3*x
```

Other examples:

```
>>> from sympy import *
>>> from sympy.abc import x, y, z
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
>>> uexpr
(5/7)*(3/4)
>>> x*UnevaluatedExpr(1/x)
x*1/x
```

A point to be noted is that `UnevaluatedExpr` cannot prevent the evaluation of an expression which is given as argument. For example:

```
>>> expr1 = UnevaluatedExpr(x + x)
>>> expr1
2*x
>>> expr2 = sympify('x + x', evaluate=False)
>>> expr2
x + x
```

Remember that `expr2` will be evaluated if included into another expression. Combine both of the methods to prevent both inside and outside evaluations:

```
>>> UnevaluatedExpr(sympify("x + x", evaluate=False)) + y
y + x + x
```

UnevaluatedExpr is supported by SymPy printers and can be used to print the result in different output forms. For example

```
>>> from sympy import latex
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
>>> print(latex(uexpr))
\frac{5}{7} \frac{3}{4}
```

In order to release the expression and get the evaluated LaTeX form, just use `.doit()`:

```
>>> print(latex(uexpr.doit()))
\frac{15}{28}
```

GOTCHAS AND PITFALLS

3.1 Introduction

SymPy runs under the [Python Programming Language](#), so there are some things that may behave differently than they do in other, independent computer algebra systems like Maple or Mathematica. These are some of the gotchas and pitfalls that you may encounter when using SymPy. See also the [FAQ](#), the [Tutorial](#) (page 3), the remainder of the SymPy Docs, and the [official Python Tutorial](#).

If you are already familiar with C or Java, you might also want to look at this [4 minute Python tutorial](#).

Ignore `#doctest: +SKIP` in the examples. That has to do with internal testing of the examples.

3.2 Equals Signs (=)

3.2.1 Single Equals Sign

The equals sign (`=`) is the assignment operator, not equality. If you want to do $x = y$, use `Eq(x, y)` for equality. Alternatively, all expressions are assumed to equal zero, so you can just subtract one side and use `x - y`.

The proper use of the equals sign is to assign expressions to variables.

For example:

```
>>> from sympy.abc import x, y
>>> a = x - y
>>> print(a)
x - y
```

3.2.2 Double Equals Signs

Double equals signs (`==`) are used to test equality. However, this tests expressions exactly, not symbolically. For example:

```
>>> (x + 1)**2 == x**2 + 2*x + 1
False
>>> (x + 1)**2 == (x + 1)**2
True
```

If you want to test for symbolic equality, one way is to subtract one expression from the other and run it through functions like `expand()`, `simplify()`, and `trigsimp()` and see if the equation reduces to 0.

```
>>> from sympy import simplify, cos, sin, expand
>>> simplify((x + 1)**2 - (x**2 + 2*x + 1))
0
>>> eq = sin(2*x) - 2*sin(x)*cos(x)
>>> simplify(eq)
0
>>> expand(eq, trig=True)
0
```

Note: See also [Why does SymPy say that two equal expressions are unequal?](#) in the FAQ.

3.3 Variables

3.3.1 Variables Assignment does not Create a Relation Between Expressions

When you use `=` to do assignment, remember that in Python, as in most programming languages, the variable does not change if you change the value you assigned to it. The equations you are typing use the values present at the time of creation to “fill in” values, just like regular Python definitions. They are not altered by changes made afterwards. Consider the following:

```
>>> from sympy import Symbol
>>> a = Symbol('a') # Symbol, 'a', stored as variable "a"
>>> b = a + 1       # an expression involving 'a' stored as variable "b"
>>> print(b)
a + 1
>>> a = 4           # "a" now points to literal integer 4, not Symbol('a')
>>> print(a)
4
>>> print(b)       # "b" is still pointing at the expression involving 'a'
a + 1
```

Changing quantity `a` does not change `b`; you are not working with a set of simultaneous equations. It might be helpful to remember that the string that gets printed when you print a variable referring to a SymPy object is the string that was given to it when it was created; that string does not have to be the same as the variable that you assign it to.

```
>>> from sympy import var
>>> r, t, d = var('rate time short_life')
>>> d = r*t
>>> print(d)
rate*time
>>> r = 80
>>> t = 2
>>> print(d)      # We haven't changed d, only r and t
rate*time
>>> d = r*t
```

(continues on next page)

(continued from previous page)

```
>>> print(d)           # Now d is using the current values of r and t
160
```

If you need variables that have dependence on each other, you can define functions. Use the `def` operator. Indent the body of the function. See the Python docs for more information on defining functions.

```
>>> c, d = var('c d')
>>> print(c)
c
>>> print(d)
d
>>> def ctimesd():
...     """
...     This function returns whatever c is times whatever d is.
...     """
...     return c*d
>>> ctimesd()
c*d
>>> c = 2
>>> print(c)
2
>>> ctimesd()
2*d
```

If you define a circular relationship, you will get a `RuntimeError`.

```
>>> def a():
...     return b()
...
>>> def b():
...     return a()
...
>>> a()
Traceback (most recent call last):
  File "...", line ..., in ...
    compileflags, 1) in test.globs
  File "<...>", line 1, in <module>
    a()
  File "<...>", line 2, in a
    return b()
  File "<...>", line 2, in b
    return a()
  File "<...>", line 2, in a
    return b()
...
RuntimeError: maximum recursion depth exceeded
```

Note: See also [Why doesn't changing one variable change another that depends on it?](#) in the FAQ.

3.3.2 Symbols

Symbols are variables, and like all other variables, they need to be assigned before you can use them. For example:

```
>>> import sympy
>>> z**2 # z is not defined yet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> sympy.var('z') # This is the easiest way to define z as a standard symbol
z
>>> z**2
z**2
```

If you use **isympy**, it runs the following commands for you, giving you some default Symbols and Functions.

```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

You can also import common symbol names from `sympy.abc`.

```
>>> from sympy.abc import w
>>> w
w
>>> import sympy
>>> dir(sympy.abc)
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
 'P', 'Q', 'R', 'S', 'Symbol', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
 '_builtins', '_doc_', '_file_', '_name_', '_package_', '_greek',
 '_latin', 'a', 'alpha', 'b', 'beta', 'c', 'chi', 'd', 'delta', 'e',
 'epsilon', 'eta', 'f', 'g', 'gamma', 'h', 'i', 'iota', 'j', 'k', 'kappa',
 'l', 'm', 'mu', 'n', 'nu', 'o', 'omega', 'omicron', 'p', 'phi', 'pi',
 'psi', 'q', 'r', 'rho', 's', 'sigma', 't', 'tau', 'theta', 'u', 'upsilon',
 'v', 'w', 'x', 'xi', 'y', 'z', 'zeta']
```

If you want control over the assumptions of the variables, use `Symbol()` and `symbols()`. See [Keyword Arguments](#) (page 75) below.

Lastly, it is recommended that you not use I, E, S, N, C, O, or Q for variable or symbol names, as those are used for the imaginary unit (*i*), the base of the natural logarithm (*e*), the `sympify()` function (see [Symbolic Expressions](#) (page 69) below), numeric evaluation (`N()` is equivalent to `evalf()` (page 374)), the big O order symbol (as in $O(n \log n)$), and the assumptions object that holds a list of supported ask keys (such as `Q.real`), respectively. You can use the mnemonic OSINEQ to remember what Symbols are defined by default in SymPy. Or better yet, always use lowercase letters for Symbol names. Python will not prevent you from overriding default SymPy names or functions, so be careful.

```
>>> cos(pi) # cos and pi are a built-in sympy names.
-1
>>> pi = 3 # Notice that there is no warning for overriding pi.
>>> cos(pi)
cos(3)
```

(continues on next page)

(continued from previous page)

```
>>> def cos(x): # No warning for overriding built-in functions either.
...     return 5*x
...
>>> cos(pi)
15
>>> from sympy import cos # reimport to restore normal behavior
```

To get a full list of all default names in SymPy do:

```
>>> import sympy
>>> dir(sympy)
# A big list of all default sympy names and functions follows.
# Ignore everything that starts and ends with __.
```

If you have `IPython` installed and use `isympy`, you can also press the TAB key to get a list of all built-in names and to autocomplete. Also, see [this page](#) for a trick for getting tab completion in the regular Python console.

Note: See also [What is the best way to create symbols?](#) in the FAQ.

3.4 Symbolic Expressions

3.4.1 Python numbers vs. SymPy Numbers

SymPy uses its own classes for integers, rational numbers, and floating point numbers instead of the default Python `int` and `float` types because it allows for more control. But you have to be careful. If you type an expression that just has numbers in it, it will default to a Python expression. Use the `sympify()` function, or just `S()`, to ensure that something is a SymPy expression.

```
>>> 6.2 # Python float. Notice the floating point accuracy problems.
6.2000000000000002
>>> type(6.2) # <type 'float'> in Python 2.x, <class 'float'> in Py3k
<... 'float'>
>>> S(6.2) # SymPy Float has no such problems because of arbitrary precision.
6.200000000000000
>>> type(S(6.2))
<class 'sympy.core.numbers.Float'>
```

If you include numbers in a SymPy expression, they will be sympified automatically, but there is one gotcha you should be aware of. If you do `<number>/<number>` inside of a SymPy expression, Python will evaluate the two numbers before SymPy has a chance to get to them. The solution is to `sympify()` one of the numbers, or use `Rational`.

```
>>> x**(1/2) # evaluates to x**0 or x**0.5
x**0.5
>>> x**(S(1)/2) # sympify one of the ints
sqrt(x)
>>> x**Rational(1, 2) # use the Rational class
sqrt(x)
```

With a power of `1/2` you can also use `sqrt` shorthand:

```
>>> sqrt(x) == x**Rational(1, 2)
True
```

If the two integers are not directly separated by a division sign then you don't have to worry about this problem:

```
>>> x**(2*x/3)
x**(2*x/3)
```

Note: A common mistake is copying an expression that is printed and reusing it. If the expression has a Rational (i.e., <number>/<number>) in it, you will not get the same result, obtaining the Python result for the division rather than a SymPy Rational.

```
>>> x = Symbol('x')
>>> print(solve(7*x - 22, x))
[22/7]
>>> 22/7 # If we just copy and paste we get int 3 or a float
3.142857142857143
>>> # One solution is to just assign the expression to a variable
>>> # if we need to use it again.
>>> a = solve(7*x - 22, x)[0]
>>> a
22/7
```

The other solution is to put quotes around the expression and run it through S() (i.e., sympify it):

```
>>> S("22/7")
22/7
```

Also, if you do not use **isypy**, you could use from `__future__` import `division` to prevent the / sign from performing integer division.

```
>>> from __future__ import division
>>> 1/2 # With division imported it evaluates to a python float
0.5
>>> 1//2 # You can still achieve integer division with //
0
```

But be careful: you will now receive floats where you might have desired a Rational:

```
>>> x**(1/2)
x**0.5
```

Rational only works for number/number and is only meant for rational numbers. If you want a fraction with symbols or expressions in it, just use /. If you do number/expression or expression/number, then the number will automatically be converted into a SymPy Number. You only need to be careful with number/number.

```
>>> Rational(2, x)
Traceback (most recent call last):
...
TypeError: invalid input: x
>>> 2/x
2/x
```

3.4.2 Evaluating Expressions with Floats and Rationals

SymPy keeps track of the precision of Float objects. The default precision is 15 digits. When an expression involving a Float is evaluated, the result will be expressed to 15 digits of precision but those digits (depending on the numbers involved with the calculation) may not all be significant.

The first issue to keep in mind is how the Float is created: it is created with a value and a precision. The precision indicates how precise of a value to use when that Float (or an expression it appears in) is evaluated.

The values can be given as strings, integers, floats, or rationals.

- strings and integers are interpreted as exact

```
>>> Float(100)
100.0000000000000
>>> Float('100', 5)
100.00
```

- to have the precision match the number of digits, the null string can be used for the precision

```
>>> Float(100, '')
100.
>>> Float('12.34')
12.3400000000000
>>> Float('12.34', '')
12.34
```

```
>>> s, r = [Float(j, 3) for j in ('0.25', Rational(1, 7))]
>>> for f in [s, r]:
...     print(f)
0.250
0.143
```

Next, notice that each of those values looks correct to 3 digits. But if we try to evaluate them to 20 digits, a difference will become apparent:

The 0.25 (with precision of 3) represents a number that has a non-repeating binary decimal; 1/7 is repeating in binary and decimal - it cannot be represented accurately too far past those first 3 digits (the correct decimal is a repeating 142857):

```
>>> s.n(20)
0.25000000000000000000
>>> r.n(20)
0.14285278320312500000
```

It is important to realize that although a Float is being displayed in decimal at arbitrary precision, it is actually stored in binary. Once the Float is created, its binary information is set at the given precision. The accuracy of that value cannot be subsequently changed; so 1/7, at a precision of 3 digits, can be padded with binary zeros, but these will not make it a more accurate value of 1/7.

If inexact, low-precision numbers are involved in a calculation with higher precision values, the evalf engine will increase the precision of the low precision values and inexact results will be obtained. This is feature of calculations with limited precision:

```
>>> Float('0.1', 10) + Float('0.1', 3)
0.2000061035
```

Although the `evalf` engine tried to maintain 10 digits of precision (since that was the highest precision represented) the 3-digit precision used limits the accuracy to about 4 digits – not all the digits you see are significant. `evalf` doesn't try to keep track of the number of significant digits.

That very simple expression involving the addition of two numbers with different precisions will hopefully be instructive in helping you understand why more complicated expressions (like trig expressions that may not be simplified) will not evaluate to an exact zero even though, with the right simplification, they should be zero. Consider this unsimplified trig identity, multiplied by a big number:

```
>>> big = 12345678901234567890
>>> big_trig_identity = big*cos(x)**2 + big*sin(x)**2 - big*1
>>> abs(big_trig_identity.subs(x, .1).n(2)) > 1000
True
```

When the `cos` and `sin` terms were evaluated to 15 digits of precision and multiplied by the big number, they gave a large number that was only precise to 15 digits (approximately) and when the 20 digit big number was subtracted the result was not zero.

There are three things that will help you obtain more precise numerical values for expressions:

- 1) Pass the desired substitutions with the call to `evaluate`. By doing the `subs` first, the `Float` values cannot be updated as necessary. By passing the desired substitutions with the call to `evalf` the ability to re-evaluate as necessary is gained and the results are impressively better:

```
>>> big_trig_identity.n(2, {x: 0.1})
-0.e-91
```

- 2) Use `Rationals`, not `Floats`. During the evaluation process, the `Rational` can be computed to an arbitrary precision while the `Float`, once created – at a default of 15 digits – cannot. Compare the value of `-1.4e+3` above with the nearly zero value obtained when replacing `x` with a `Rational` representing `1/10` – before the call to `evaluate`:

```
>>> big_trig_identity.subs(x, S('1/10')).n(2)
0.e-91
```

- 3) Try to simplify the expression. In this case, SymPy will recognize the trig identity and simplify it to zero so you don't even have to evaluate it numerically:

```
>>> big_trig_identity.simplify()
0
```

3.4.3 Immutability of Expressions

Expressions in SymPy are immutable, and cannot be modified by an in-place operation. This means that a function will always return an object, and the original expression will not be modified. The following example snippet demonstrates how this works:

```
def main():
    var('x y a b')
    expr = 3*x + 4*y
    print('original =', expr)
    expr_modified = expr.subs({x: a, y: b})
    print('modified =', expr_modified)

if __name__ == "__main__":
    main()
```

The output shows that the `subs()` function has replaced variable `x` with variable `a`, and variable `y` with variable `b`:

```
original = 3*x + 4*y
modified = 3*a + 4*b
```

The `subs()` function does not modify the original expression `expr`. Rather, a modified copy of the expression is returned. This returned object is stored in the variable `expr_modified`. Note that unlike C/C++ and other high-level languages, Python does not require you to declare a variable before it is used.

3.4.4 Mathematical Operators

SymPy uses the same default operators as Python. Most of these, like `*/+-`, are standard. Aside from integer division discussed in [Python numbers vs. SymPy Numbers](#) (page 69) above, you should also be aware that implied multiplication is not allowed. You need to use `*` whenever you wish to multiply something. Also, to raise something to a power, use `**`, not `^` as many computer algebra systems use. Parentheses `()` change operator precedence as you would normally expect.

In **isympy**, with the **ipython** shell:

```
>>> 2x
Traceback (most recent call last):
...
SyntaxError: invalid syntax
>>> 2*x
2*x
>>> (x + 1)^2 # This is not power. Use ** instead.
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ^: 'Add' and 'int'
>>> (x + 1)**2
(x + 1)**2
>>> pprint(3 - x**(2*x)/(x + 1))
      2*x
      x
- ---- + 3
  x + 1
```

3.4.5 Inverse Trig Functions

SymPy uses different names for some functions than most computer algebra systems. In particular, the inverse trig functions use the python names of `asin()`, `acos()` and so on instead

of the usual `arcsin` and `arccos`. Use the methods described in [Symbols](#) (page 68) above to see the names of all SymPy functions.

3.5 Special Symbols

The symbols `[]`, `{}`, `=`, and `()` have special meanings in Python, and thus in SymPy. See the Python docs linked to above for additional information.

3.5.1 Lists

Square brackets `[]` denote a list. A list is a container that holds any number of different objects. A list can contain anything, including items of different types. Lists are mutable, which means that you can change the elements of a list after it has been created. You access the items of a list also using square brackets, placing them after the list or list variable. Items are numbered using the space before the item.

Note: List indexes begin at 0.

Example:

```
>>> a = [x, 1] # A simple list of two items
>>> a
[x, 1]
>>> a[0] # This is the first item
x
>>> a[0] = 2 # You can change values of lists after they have been created
>>> print(a)
[2, 1]
>>> print(solve(x**2 + 2*x - 1, x)) # Some functions return lists
[-1 + sqrt(2), -sqrt(2) - 1]
```

Note: See the Python docs for more information on lists and the square bracket notation for accessing elements of a list.

3.5.2 Dictionaries

Curly brackets `{}` denote a dictionary, or a dict for short. A dictionary is an unordered list of non-duplicate keys and values. The syntax is `{key: value}`. You can access values of keys using square bracket notation.

```
>>> d = {'a': 1, 'b': 2} # A dictionary.
>>> d
{'a': 1, 'b': 2}
>>> d['a'] # How to access items in a dict
1
>>> roots((x - 1)**2*(x - 2), x) # Some functions return dicts
{1: 2, 2: 1}
>>> # Some SymPy functions return dictionaries. For example,
```

(continues on next page)

(continued from previous page)

```
>>> # roots returns a dictionary of root:multiplicity items.
>>> roots((x - 5)**2*(x + 3), x)
{-3: 1, 5: 2}
>>> # This means that the root -3 occurs once and the root 5 occurs twice.
```

Note: See the Python docs for more information on dictionaries.

3.5.3 Tuples

Parentheses `()`, aside from changing operator precedence and their use in function calls, (like `cos(x)`), are also used for tuples. A tuple is identical to a [list](#) (page 74), except that it is not mutable. That means that you cannot change their values after they have been created. In general, you will not need tuples in SymPy, but sometimes it can be more convenient to type parentheses instead of square brackets.

```
>>> t = (1, 2, x) # Tuples are like lists
>>> t
(1, 2, x)
>>> t[0]
1
>>> t[0] = 4 # Except you cannot change them after they have been created
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Single element tuples, unlike lists, must have a comma in them:

```
>>> (x,)
(x,)
```

Without the comma, a single expression without a comma is not a tuple:

```
>>> (x)
x
```

`integrate` takes a sequence as the second argument if you want to integrate with limits (and a tuple or list will work):

```
>>> integrate(x**2, (x, 0, 1))
1/3
>>> integrate(x**2, [x, 0, 1])
1/3
```

Note: See the Python docs for more information on tuples.

3.5.4 Keyword Arguments

Aside from the usage described [above](#) (page 65), equals signs (`=`) are also used to give named arguments to functions. Any function that has `key=value` in its parameters list (see below on how to find this out), then `key` is set to `value` by default. You can change the value of the

key by supplying your own value using the equals sign in the function call. Also, functions that have `**` followed by a name in the parameters list (usually `**kwargs` or `**assumptions`) allow you to add any number of key=value pairs that you want, and they will all be evaluated according to the function.

`sqrt(x**2)` doesn't auto simplify to `x` because `x` is assumed to be complex by default, and, for example, `sqrt((-1)**2) == sqrt(1) == 1 != -1`:

```
>>> sqrt(x**2)
sqrt(x**2)
```

Giving assumptions to Symbols is an example of using the keyword argument:

```
>>> x = Symbol('x', positive=True)
```

The square root will now simplify since it knows that `x >= 0`:

```
>>> sqrt(x**2)
x
```

`powsimp` has a default argument of `combine='all'`:

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m))
      m + n
(x*y)
```

Setting `combine` to the default value is the same as not setting it.

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='all'))
      m + n
(x*y)
```

The non-default options are `'exp'`, which combines exponents...

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='exp'))
      m + n   m + n
x      *y
```

...and `'base'`, which combines bases.

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='base'))
      m      n
(x*y) *(x*y)
```

Note: See the Python docs for more information on function parameters.

3.6 Getting help from within SymPy

3.6.1 `help()`

Although all docs are available at docs.sympy.org or on the [SymPy Wiki](#), you can also get info on functions from within the Python interpreter that runs SymPy. The easiest way to do this is to do `help(function)`, or `function?` if you are using **ipython**:

```
In [1]: help(powsimp) # help() works everywhere
```

```
In [2]: # But in ipython, you can also use ?, which is better because it
```

```
In [3]: # it gives you more information
```

```
In [4]: powsimp?
```

These will give you the function parameters and docstring for `powsimp()`. The output will look something like this:

```
sympy.simplify.simplify.powsimp(expr, deep=False, combine='all', force=False,
                                measure=<function count_ops>)
    reduces expression by combining powers with similar bases and exponents.
```

Notes

If `deep` is `True` then `powsimp()` will also simplify arguments of functions. By default `deep` is set to `False`.

If `force` is `True` then bases will be combined without checking for assumptions, e.g. $\sqrt{x} \cdot \sqrt{y} \rightarrow \sqrt{x \cdot y}$ which is not true if x and y are both negative.

You can make `powsimp()` only combine bases or only combine exponents by changing `combine='base'` or `combine='exp'`. By default, `combine='all'`, which does both. `combine='base'` will only combine:

```

  a   a           a           2x           x
x * y => (x*y)  as well as things like 2  => 4
```

and `combine='exp'` will only combine

```

  a   b       (a + b)
x * x => x
```

`combine='exp'` will strictly only combine exponents in the way that used to be automatic. Also use `deep=True` if you need the old behavior.

When `combine='all'`, `'exp'` is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want `'base'` combined first, do something like `powsimp(powsimp(expr, combine='base'), combine='exp')`.

Examples

```
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
```

(continues on next page)

(continued from previous page)

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> from sympy import sqrt, Mul
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _*a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

3.6.2 source()

Another useful option is the `source()` function. This will print the source code of a function, including any docstring that it may have. You can also do `function??` in **ipython**. For example, from SymPy 0.6.5:

```
>>> source(simplify) # simplify() is actually only 2 lines of code.
In file: ./sympy/simplify/simplify.py
def simplify(expr):
    """Naively simplifies the given expression.

    ...
    Simplification is not a well defined term and the exact strategies
    this function tries can change in the future versions of SymPy. If
    your algorithm relies on "simplification" (whatever it is), try to
    determine what you need exactly - is it powsimp()? radsimp()?
    together()?, logcombine()?, or something else? And use this particular
    function directly, because those are well defined and thus your algorithm
    will be robust.

    ...
    """
```

(continues on next page)

(continued from previous page)

```
expr = Poly.cancel(powsimp(expr))  
return powsimp(together(expr.expand()), combine='exp', deep=True)
```


SYMPY USER'S GUIDE

4.1 Introduction

If you are new to SymPy, start with the [Tutorial](#) (page 3). If you went through it, now it's time to learn how SymPy works internally, which is what this guide is about. Once you grasp the ideas behind SymPy, you will be able to use it effectively and also know how to extend it and fix it. You may also be just interested in [SymPy Modules Reference](#) (page 91).

4.2 Learning SymPy

Everyone has different ways of understanding the code written by others.

4.2.1 Ondřej's approach

Let's say I'd like to understand how $x + y + x$ works and how it is possible that it gets simplified to $2*x + y$.

I write a simple script, I usually call it `t.py` (I don't remember anymore why I call it that way):

```
from sympy.abc import x, y
e = x + y + x
print e
```

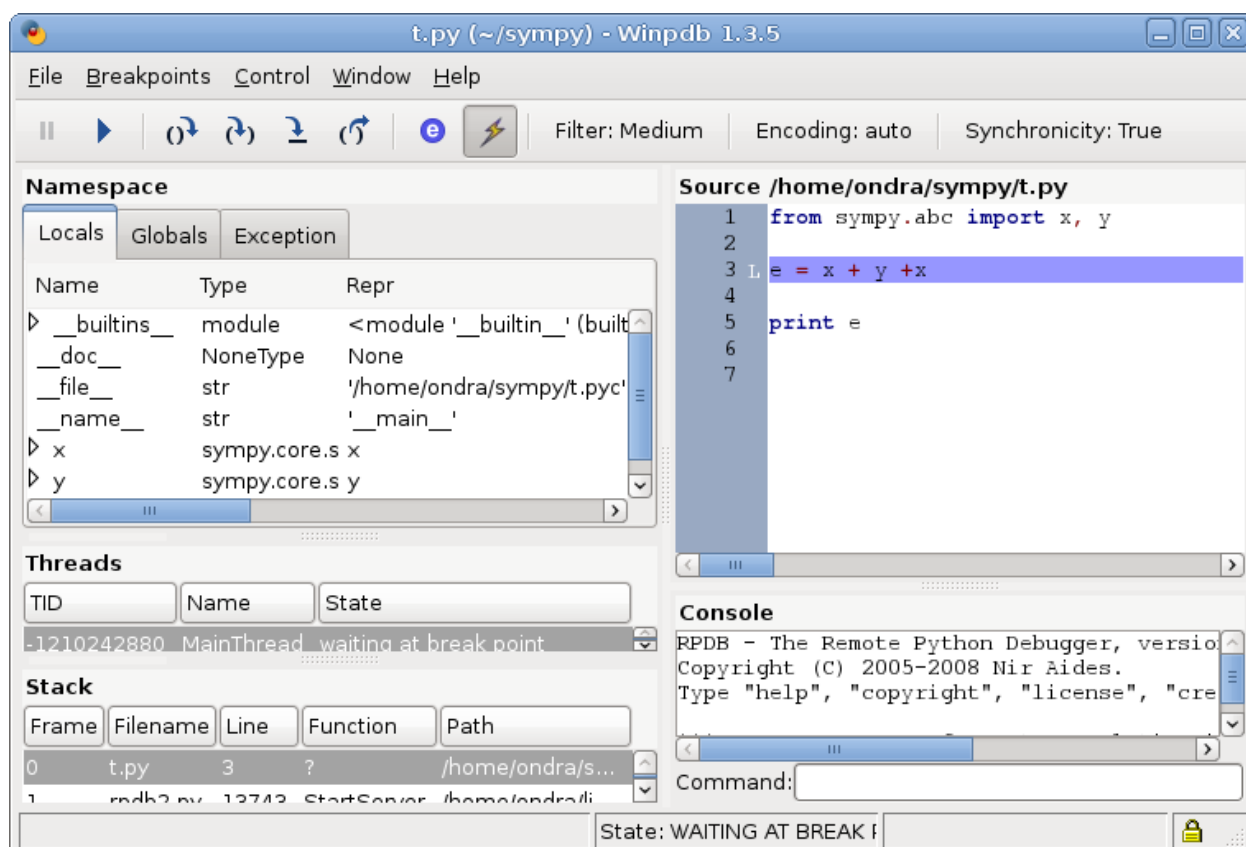
And I try if it works

```
$ python t.py
y + 2*x
```

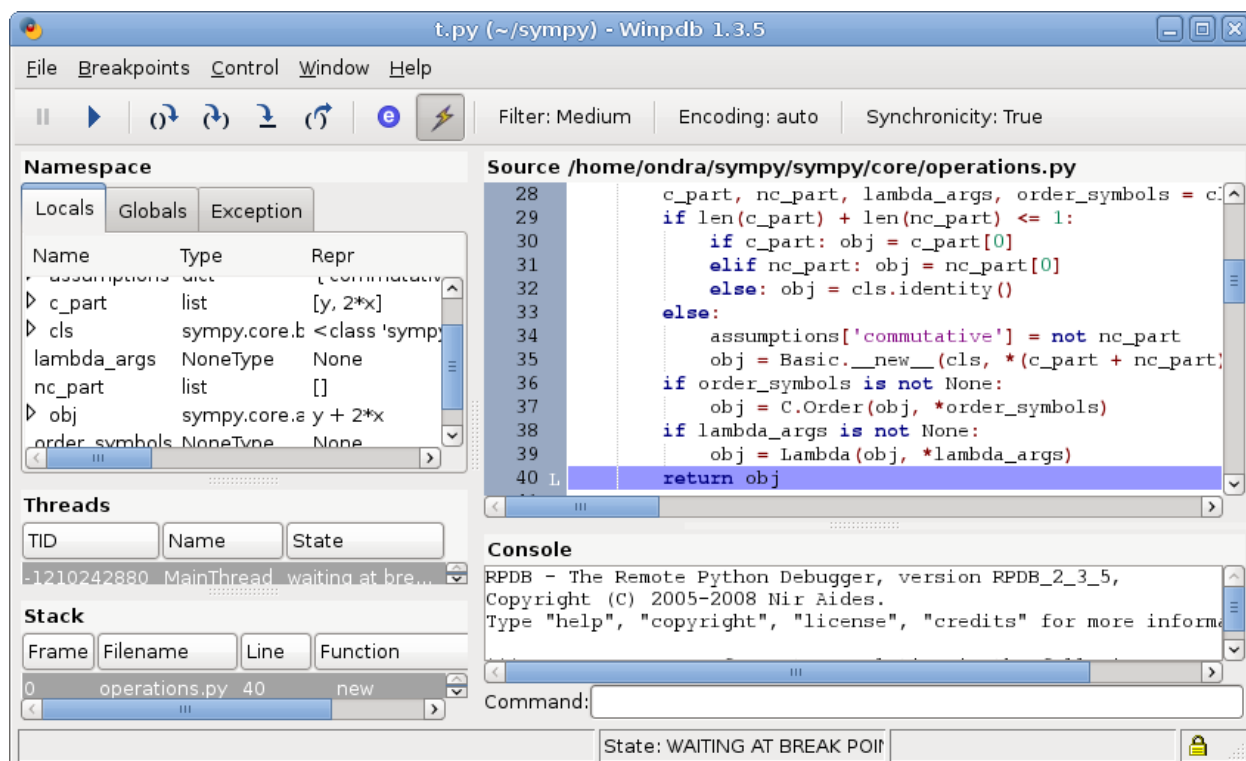
Now I start [winpdb](#) on it (if you've never used winpdb – it's an excellent multiplatform debugger, works on Linux, Windows and Mac OS X):

```
$ winpdb t.py
y + 2*x
```

and a winpdb window will popup, I move to the next line using F6:



Then I step into (F7) and after a little debugging I get for example:



Tip: Make the winpdb window larger on your screen, it was just made smaller to fit in this guide.

I see values of all local variables in the left panel, so it's very easy to see what's happening. You can see, that the $y + 2x$ is emerging in the `obj` variable. Observing that `obj` is constructed from `c_part` and `nc_part` and seeing what `c_part` contains (y and $2x$). So looking at the line 28 (the whole line is not visible on the screenshot, so here it is):

```
c_part, nc_part, lambda_args, order_symbols = cls.flatten(map(sympify, args))
```

you can see that the simplification happens in `cls.flatten`. Now you can set the breakpoint on the line 28, quit winpdb (it will remember the breakpoint), start it again, hit F5, this will stop at this breakpoint, hit F7, this will go into the function `Add.flatten()`:

```
@classmethod
def flatten(cls, seq):
    """
    Takes the sequence "seq" of nested Adds and returns a flatten list.

    Returns: (commutative_part, noncommutative_part, lambda_args,
             order_symbols)

    Applies associativity, all terms are commutable with respect to
    addition.
    """
    terms = {}          # term -> coeff
                        # e.g. x**2 -> 5   for ... + 5*x**2 + ...

    coeff = S.Zero      # standalone term
                        # e.g. 3 + ...
    lambda_args = None
    order_factors = []
    while seq:
        o = seq.pop(0)
```

and then you can study how it works. I am going to stop here, this should be enough to get you going - with the above technique, I am able to understand almost any Python code.

4.3 SymPy's Architecture

We try to make the sources easily understandable, so you can look into the sources and read the doctests, it should be well documented and if you don't understand something, ask on the [mailinglist](#).

You can find all the decisions archived in the [issues](#), to see rationale for doing this and that.

4.3.1 Basics

All symbolic things are implemented using subclasses of the `Basic` class. First, you need to create symbols using `Symbol("x")` or numbers using `Integer(5)` or `Float(34.3)`. Then you construct the expression using any class from SymPy. For example `Add(Symbol("a"), Symbol("b"))` gives an instance of the `Add` class. You can call all methods, which the particular class supports.

For easier use, there is a syntactic sugar for expressions.

For example, `cos(x) + 1` is equal to `cos(x).__add__(1)` is equal to `Add(cos(x), Integer(1))`

This works because Python allows you to override the behavior of built-in operations `+`, `*`, `-`, `/` etc., by defining methods with names `__add__`, `__mul__` etc., so `cos(x) + 1` is evaluated as `cos(x).__add__(1)`.

Then, in the `__add__`, `__mul__` methods defined for SymPy objects, new `Add` or `Mul` objects are constructed. Python types are also converted to SymPy types through the `sympify()` function. Thus, `cos(x).__add__(1)` returns `Add(cos(x), Integer(1))`.

Similarly, `2/cos(x)` is equal to `cos(x).__rdiv__(2)` is equal to `Mul(Integer(2), Pow(cos(x), Integer(-1)))`.

Note that `2/cos(x)` calls `cos(x).__rdiv__(2)` instead of `(2).__div__(cos(x))` because `2` (type `int`) does not know how to be divided by `cos(x)`. (Note: In Python 3, `__div__` is replaced by `__truediv__` and `__rdiv__` is replaced by `__rtruediv__`.)

So, you can write normal expressions using python arithmetics like this:

```
a = Symbol("a")
b = Symbol("b")
e = (a + b)**2
print e
```

but from the SymPy point of view, we just need the classes `Add`, `Mul`, `Pow`, `Rational`, `Integer`.

4.3.2 Automatic evaluation to canonical form

For computation, all expressions need to be in a canonical form, this is done during the creation of the particular instance and only inexpensive operations are performed, necessary to put the expression in the canonical form. So the canonical form doesn't mean the simplest possible expression. The exact list of operations performed depend on the implementation. Obviously, the definition of the canonical form is arbitrary, the only requirement is that all equivalent expressions must have the same canonical form. We tried the conversion to a canonical (standard) form to be as fast as possible and also in a way so that the result is what you would write by hand - so for example `b*a + -4 + b + a*b + 4 + (a + b)**2` becomes `2*a*b + b + (a + b)**2`.

Whenever you construct an expression, for example `Add(x, x)`, the `Add.__new__()` is called and it determines what to return. In this case:

```
>>> from sympy import Add
>>> from sympy.abc import x
>>> e = Add(x, x)
>>> e
2*x

>>> type(e)
<class 'sympy.core.mul.Mul'>
```

`e` is actually an instance of `Mul(2, x)`, because `Add.__new__()` returned `Mul`.

4.3.3 Comparisons

Expressions can be compared using a regular python syntax:

```
>>> from sympy.abc import x, y
>>> x + y == y + x
True

>>> x + y == y - x
False
```

We made the following decision in SymPy: `a = Symbol("x")` and another `b = Symbol("x")` (with the same string "x") is the same thing, i.e. `a == b` is `True`. We chose `a == b`, because it is more natural - `exp(x) == exp(x)` is also `True` for the same instance of `x` but different instances of `exp`, so we chose to have `exp(x) == exp(x)` even for different instances of `x`.

Sometimes, you need to have a unique symbol, for example as a temporary one in some calculation, which is going to be substituted for something else at the end anyway. This is achieved using `Dummy("x")`. So, to sum it up:

```
>>> from sympy import Symbol, Dummy
>>> Symbol("x") == Symbol("x")
True

>>> Dummy("x") == Dummy("x")
False
```

4.3.4 Debugging

Starting with 0.6.4, you can turn on/off debug messages with the environment variable `SYMPY_DEBUG`, which is expected to have the values `True` or `False`. For example, to turn on debugging, you would issue:

```
[user@localhost]: SYMPY_DEBUG=True ./bin/isympy
```

4.3.5 Functionality

There are no given requirements on classes in the library. For example, if they don't implement the `fdiff()` method and you construct an expression using such a class, then trying to use the `Basic.series()` method will raise an exception of not finding the `fdiff()` method in your class. This "duck typing" has an advantage that you just implement the functionality which you need.

You can define the `cos` class like this:

```
class cos(Function):
    pass
```

and use it like `1 + cos(x)`, but if you don't implement the `fdiff()` method, you will not be able to call `(1 + cos(x)).series()`.

The symbolic object is characterized (defined) by the things which it can do, so implementing more methods like `fdiff()`, `subs()` etc., you are creating a "shape" of the symbolic object. Useful things to implement in new classes are: `hash()` (to use the class in comparisons), `fdiff()` (to use it in series expansion), `subs()` (to use it in expressions, where some parts are being substituted) and `series()` (if the series cannot be computed using the general `Basic.series()` method). When you create a new class, don't worry about this too much - just try to

use it in your code and you will realize immediately which methods need to be implemented in each situation.

All objects in sympy are immutable - in the sense that any operation just returns a new instance (it can return the same instance only if it didn't change). This is a common mistake to change the current instance, like `self.arg = self.arg + 1` (wrong!). Use `arg = self.arg + 1; return arg` instead. The object is immutable in the sense of the symbolic expression it represents. It can modify itself to keep track of, for example, its hash. Or it can recalculate anything regarding the expression it contains. But the expression cannot be changed. So you can pass any instance to other objects, because you don't have to worry that it will change, or that this would break anything.

4.3.6 Conclusion

Above are the main ideas behind SymPy that we try to obey. The rest depends on the current implementation and may possibly change in the future. The point of all of this is that the interdependencies inside SymPy should be kept to a minimum. If one wants to add new functionality to SymPy, all that is necessary is to create a subclass of `Basic` and implement what you want.

4.3.7 Functions

How to create a new function with one variable:

```
class sign(Function):

    nargs = 1

    @classmethod
    def eval(cls, arg):
        if isinstance(arg, Basic.NaN):
            return S.NaN
        if isinstance(arg, Basic.Zero):
            return S.Zero
        if arg.is_positive:
            return S.One
        if arg.is_negative:
            return S.NegativeOne
        if isinstance(arg, Basic.Mul):
            coeff, terms = arg.as_coeff_mul()
            if not isinstance(coeff, Basic.One):
                return cls(coeff) * cls(Basic.Mul(*terms))

    is_finite = True

    def _eval_conjugate(self):
        return self

    def _eval_is_zero(self):
        return isinstance(self[0], Basic.Zero)
```

and that's it. The `_eval_*` functions are called when something is needed. The `eval` is called when the class is about to be instantiated and it should return either some simplified instance of some other class or if the class should be unmodified, return `None` (see `core/function.py` in `Function.__new__` for implementation details). See also tests in

`sympy/functions/elementary/tests/test_interface.py` that test this interface. You can use them to create your own new functions.

The applied function `sign(x)` is constructed using

```
sign(x)
```

both inside and outside of SymPy. Unapplied functions `sign` is just the class itself:

```
sign
```

both inside and outside of SymPy. This is the current structure of classes in SymPy:

```
class BasicType(type):
    pass
class MetaBasicMeths(BasicType):
    ...
class BasicMeths(AssumeMeths):
    __metaclass__ = MetaBasicMeths
    ...
class Basic(BasicMeths):
    ...
class FunctionClass(MetaBasicMeths):
    ...
class Function(Basic, RelMeths, ArithMeths):
    __metaclass__ = FunctionClass
    ...
```

The exact names of the classes and the names of the methods and how they work can be changed in the future.

This is how to create a function with two variables:

```
class chebyshevt_root(Function):
    nargs = 2

    @classmethod
    def eval(cls, n, k):
        if not 0 <= k < n:
            raise ValueError("must have 0 <= k < n")
        return cos(S.Pi*(2*k + 1)/(2*n))
```

Note: the first argument of a `@classmethod` should be `cls` (i.e. not `self`).

Here it's how to define a derivative of the function:

```
>>> from sympy import Function, sympify, cos
>>> class my_function(Function):
...     nargs = 1
...
...     def fdiff(self, argindex = 1):
...         return cos(self.args[0])
...
...     @classmethod
...     def eval(cls, arg):
...         arg = sympify(arg)
...         if arg == 0:
...             return sympify(0)
```

So guess what this `my_function` is going to be? Well, its derivative is `cos` and the function value at 0 is 0, but let's pretend we don't know:

```
>>> from sympy import pprint
>>> pprint(my_function(x).series(x, 0, 10))
      3      5      7      9
      x      x      x      x      / 10\
x - -- + --- - ---- + ----- + 0\ x /
      6     120    5040   362880
```

Looks familiar indeed:

```
>>> from sympy import sin
>>> pprint(sin(x).series(x, 0, 10))
      3      5      7      9
      x      x      x      x      / 10\
x - -- + --- - ---- + ----- + 0\ x /
      6     120    5040   362880
```

Let's try a more complicated example. Let's define the derivative in terms of the function itself:

```
>>> class what_am_i(Function):
...     nargs = 1
...
...     def fdiff(self, argindex = 1):
...         return 1 - what_am_i(self.args[0])**2
...
...     @classmethod
...     def eval(cls, arg):
...         arg = sympify(arg)
...         if arg == 0:
...             return sympify(0)
```

So what is `what_am_i`? Let's try it:

```
>>> pprint(what_am_i(x).series(x, 0, 10))
      3      5      7      9
      x    2*x    17*x    62*x    / 10\
x - -- + --- - ---- + ----- + 0\ x /
      3     15    315    2835
```

Well, it's `tanh`:

```
>>> from sympy import tanh
>>> pprint(tanh(x).series(x, 0, 10))
      3      5      7      9
      x    2*x    17*x    62*x    / 10\
x - -- + --- - ---- + ----- + 0\ x /
      3     15    315    2835
```

The new functions we just defined are regular SymPy objects, you can use them all over SymPy, e.g.:

```
>>> from sympy import limit
>>> limit(what_am_i(x)/x, x, 0)
1
```

4.3.8 Common tasks

Please use the same way as is shown below all across SymPy.

accessing parameters:

```
>>> from sympy import sign, sin
>>> from sympy.abc import x, y, z
```

```
>>> e = sign(x**2)
>>> e.args
(x**2,)
```

```
>>> e.args[0]
x**2
```

Number arguments (in Adds and Muls) will always be the first argument; other arguments might be in arbitrary order:

```
>>> (1 + x + y*z).args[0]
1
>>> (1 + x + y*z).args[1] in (x, y*z)
True
```

```
>>> (y*z).args
(y, z)
```

```
>>> sin(y*z).args
(y*z,)
```

Never use internal methods or variables, prefixed with “_” (example: don’t use `_args`, use `.args` instead).

testing the structure of a SymPy expression

Applied functions:

```
>>> from sympy import sign, exp, Function
>>> e = sign(x**2)

>>> isinstance(e, sign)
True

>>> isinstance(e, exp)
False

>>> isinstance(e, Function)
True
```

So `e` is a `sign(z)` function, but not `exp(z)` function.

Unapplied functions:

```
>>> from sympy import sign, exp, FunctionClass
>>> e = sign

>>> f = exp

>>> g = Add
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(e, FunctionClass)
True

>>> isinstance(f, FunctionClass)
True

>>> isinstance(g, FunctionClass)
False

>>> g is Add
True
```

So `e` and `f` are functions, `g` is not a function.

4.4 Contributing

We welcome every SymPy user to participate in it's development. Don't worry if you've never contributed to any open source project, we'll help you learn anything necessary, just ask on our [mailinglist](#).

Don't be afraid to ask anything and don't worry that you are wasting our time if you are new to SymPy and ask questions that maybe most of the people know the answer to – you are not, because that's exactly what the [mailinglist](#) is for and people answer your emails because they want to. Also we try hard to answer every email, so you'll always get some feedback and pointers what to do next.

4.4.1 Improving the code

Go to [issues](#) that are sorted by priority and simply find something that you would like to get fixed and fix it. If you find something odd, please report it into [issues](#) first before fixing it. Feel free to consult with us on the [mailinglist](#). Then send your patch either to the [issues](#) or the [mailinglist](#).

Please read our excellent [SymPy Patches Tutorial](#) at our wiki for a guide on how to write patches to SymPy, how to work with Git, and how to make your life easier as you get started with SymPy.

4.4.2 Improving the docs

Please see [the documentation](#) (page 91) how to fix and improve SymPy's documentation. All contribution is very welcome.

SYMPY MODULES REFERENCE

Because every feature of SymPy must have a test case, when you are not sure how to use something, just look into the `tests/` directories, find that feature and read the tests for it, that will tell you everything you need to know.

Most of the things are already documented though in this document, that is automatically generated using SymPy's docstrings.

Click the "modules" (modindex) link in the top right corner to easily access any SymPy module, or use the list below:

5.1 SymPy Core

5.1.1 sympify

sympify

`sympy.core.sympify.sympify(a, locals=None, convert_xor=True, strict=False, rational=False, evaluate=None)`

Converts an arbitrary expression to a type that can be used inside SymPy.

For example, it will convert Python ints into instances of `sympy.Integer`, floats into instances of `sympy.Float`, etc. It is also able to coerce symbolic expressions which inherit from `Basic`. This can be useful in cooperation with SAGE.

It currently accepts as arguments:

- any object defined in sympy
- standard numeric python types: int, long, float, Decimal
- strings (like "0.09" or "2e-19")
- booleans, including None (will leave None unchanged)
- lists, sets or tuples containing any of the above

Warning: Note that this function uses `eval`, and thus shouldn't be used on unsanitized input.

If the argument is already a type that SymPy understands, it will do nothing but return that value. This can be used at the beginning of a function to ensure you are working with the correct type.

```
>>> from sympy import sympify
```

```
>>> sympify(2).is_integer
True
>>> sympify(2).is_real
True
```

```
>>> sympify(2.0).is_real
True
>>> sympify("2.0").is_real
True
>>> sympify("2e-45").is_real
True
```

If the expression could not be converted, a `SympifyError` is raised.

```
>>> sympify("x**2")
Traceback (most recent call last):
...
SympifyError: SympifyError: "could not parse u'x**2'"
```

Notes

Sometimes autosimplification during sympification results in expressions that are very different in structure than what was entered. Until such autosimplification is no longer done, the `kernS` function might be of some use. In the example below you can see how an expression reduces to -1 by autosimplification, but does not do so when `kernS` is used.

```
>>> from sympy.core.sympify import kernS
>>> from sympy.abc import x
>>> -2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
-1
>>> s = '-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1'
>>> sympify(s)
-1
>>> kernS(s)
-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
```

Locals

The sympification happens with access to everything that is loaded by `from sympy import *`; anything used in a string that is not defined by that import will be converted to a symbol. In the following, the `bitcount` function is treated as a symbol and the `0` is interpreted as the `Order` object (used with series) and it raises an error when used improperly:

```
>>> s = 'bitcount(42)'
>>> sympify(s)
bitcount(42)
>>> sympify("0(x)")
0(x)
>>> sympify("0 + 1")
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: unbound method...
```

In order to have `bitcount` be recognized it can be imported into a namespace dictionary and passed as locals:

```
>>> from sympy.core.compatibility import exec_
>>> ns = {}
>>> exec_('from sympy.core.evalf import bitcount', ns)
>>> sympify(s, locals=ns)
6
```

In order to have the 0 interpreted as a Symbol, identify it as such in the namespace dictionary. This can be done in a variety of ways; all three of the following are possibilities:

```
>>> from sympy import Symbol
>>> ns["0"] = Symbol("0") # method 1
>>> exec_('from sympy.abc import 0', ns) # method 2
>>> ns.update(dict(0=Symbol("0"))) # method 3
>>> sympify("0 + 1", locals=ns)
0 + 1
```

If you want all single-letter and Greek-letter variables to be symbols then you can use the clashing-symbols dictionaries that have been defined there as private variables: `_clash1` (single-letter variables), `_clash2` (the multi-letter Greek names) or `_clash` (both single and multi-letter names that are defined in `abc`).

```
>>> from sympy.abc import _clash1
>>> _clash1
{'C': C, 'E': E, 'I': I, 'N': N, 'O': O, 'Q': Q, 'S': S}
>>> sympify('I & Q', _clash1)
I & Q
```

Strict

If the option `strict` is set to `True`, only the types for which an explicit conversion has been defined are converted. In the other cases, a `SympifyError` is raised.

```
>>> print(sympify(None))
None
>>> sympify(None, strict=True)
Traceback (most recent call last):
...
SympifyError: SympifyError: None
```

Evaluation

If the option `evaluate` is set to `False`, then arithmetic and operators will be converted into their SymPy equivalents and the `evaluate=False` option will be added. Nested Add or Mul will be denested first. This is done via an AST transformation that replaces operators with their SymPy equivalents, so if an operand redefines any of those operations, the redefined operators will not be used.

```
>>> sympify('2**2 / 3 + 5')
19/3
>>> sympify('2**2 / 3 + 5', evaluate=False)
2**2/3 + 5
```

Extending

To extend sympify to convert custom objects (not derived from Basic), just define a `_sympy_` method to your class. You can do that even to classes that you do not own by subclassing or adding the method at runtime.

```
>>> from sympy import Matrix
>>> class MyList1(object):
...     def __iter__(self):
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i): return list(self)[i]
...     def _sympy_(self): return Matrix(self)
>>> sympify(MyList1())
Matrix([
[1],
[2]])
```

If you do not have control over the class definition you could also use the converter global dictionary. The key is the class and the value is a function that takes a single argument and returns the desired SymPy object, e.g. `converter[MyList] = lambda x: Matrix(x)`.

```
>>> class MyList2(object): # XXX Do not do this if you control the class!
...     def __iter__(self): # Use _sympy_!
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i): return list(self)[i]
>>> from sympy.core.sympify import converter
>>> converter[MyList2] = lambda x: Matrix(x)
>>> sympify(MyList2())
Matrix([
[1],
[2]])
```

5.1.2 assumptions

This module contains the machinery handling assumptions.

All symbolic objects have assumption attributes that can be accessed via `.is_<assumption name>` attribute.

Assumptions determine certain properties of symbolic objects and can have 3 possible values: True, False, None. True is returned if the object has the property and False is returned if it doesn't or can't (i.e. doesn't make sense):

```
>>> from sympy import I
>>> I.is_algebraic
True
>>> I.is_real
False
>>> I.is_prime
False
```

When the property cannot be determined (or when a method is not implemented) None will be returned, e.g. a generic symbol, x , may or may not be positive so a value of None is returned for `x.is_positive`.

By default, all symbolic values are in the largest set in the given context without specifying the property. For example, a symbol that has a property being integer, is also real, complex, etc.

Here follows a list of possible assumption names:

commutative object commutes with any other object with respect to multiplication operation.

complex object can have only values from the set of complex numbers.

imaginary object value is a number that can be written as a real number multiplied by the imaginary unit I . See [R51] (page 1853). Please note, that 0 is not considered to be an imaginary number, see [issue #7649](#).

real object can have only values from the set of real numbers.

integer object can have only values from the set of integers.

odd

even object can have only values from the set of odd (even) integers [R50] (page 1853).

prime object is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [R54] (page 1853).

composite object is a positive integer that has at least one positive divisor other than 1 or the number itself. See [R52] (page 1853).

zero object has the value of 0.

nonzero object is a real number that is not zero.

rational object can have only values from the set of rationals.

algebraic object can have only values from the set of algebraic numbers¹¹.

transcendental object can have only values from the set of transcendental numbers¹⁰.

irrational object value cannot be represented exactly by Rational, see [R53] (page 1853).

finite

infinite object absolute value is bounded (arbitrarily large). See [R55] (page 1853), [R56] (page 1853), [R57] (page 1853).

negative

nonnegative object can have only negative (nonnegative) values [R49] (page 1853).

positive

nonpositive object can have only positive (only nonpositive) values.

¹¹ http://en.wikipedia.org/wiki/Algebraic_number

¹⁰ http://en.wikipedia.org/wiki/Transcendental_number

hermitian

antihermitian object belongs to the field of hermitian (antihermitian) operators.

Examples

```
>>> from sympy import Symbol
>>> x = Symbol('x', real=True); x
x
>>> x.is_real
True
>>> x.is_complex
True
```

See Also

See also:

[sympy.core.numbers.ImaginaryUnit](#) (page 149) [sympy.core.numbers.Zero](#) (page 144)
[sympy.core.numbers.One](#) (page 145)

Notes

Assumption values are stored in `obj._assumptions` dictionary or are returned by getter methods (with property decorators) or are attributes of objects/classes.

References

5.1.3 cache

cacheit

`sympy.core.cache.cacheit(func)`

5.1.4 basic

Basic

class `sympy.core.basic.Basic`

Base class for all objects in SymPy.

Conventions:

1. Always use `.args`, when accessing parameters of some instance:

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

2. Never use internal methods or variables (the ones prefixed with `_`):

```
>>> cot(x)._args      # do not use this, use cot(x).args instead
(x,)
```

args

Returns a tuple of arguments of 'self'.

Notes

Never use `self._args`, always use `self.args`. Only use `_args` in `__new__` when creating a new function. Don't override `.args()` from Basic (so that it's easy to change the interface in the future if needed).

Examples

```
>>> from sympy import cot
>>> from sympy.abc import x, y
```

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

as_content_primitive(radical=False, clear=True)

A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.

See also:

[sympy.core.expr.Expr.as_content_primitive](#) (page 114)

as_poly(*gens, **args)

Converts self to a polynomial or returns None.

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> print((x**2 + x*y).as_poly())
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> print((x**2 + x*y).as_poly(x, y))
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> print((x**2 + sin(y)).as_poly(x, y))
None
```

assumptions0

Return object *type* assumptions.

For example:

```
Symbol('x', real=True) Symbol('x', integer=True)
```

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'hermitian': True,
'imaginary': False, 'negative': False, 'nonnegative': True,
'nonpositive': False, 'nonzero': True, 'positive': True, 'real': True,
'zero': False}
```

atoms(*types)

Returns the atoms that form the current object.

By default, only objects that are truly atomic and can't be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and pi. It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
```



```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that `I` (imaginary unit) and `zoo` (complex infinity) are special types of number symbols and are not part of the `NumberSymbol` class.

The type can be given implicitly, too:

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}
```

Be careful to check your assumptions when using the implicit option since `S(1)`.
`is_Integer = True` but `type(S(1))` is `One`, a special type of sympy atom, while `type(S(2))` is type `Integer` and will find all integers in an expression:

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}
```

Finally, arguments to `atoms()` can select more than atomic atoms: any sympy type (loaded in `core/_init_.py`) can be listed as an argument and those types of “atoms” as found in scanning the arguments of the expression recursively:

```
>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

canonical_variables

Return a dictionary mapping any variable defined in `self.variables` as underscore-suffixed numbers corresponding to their position in `self.variables`. Enough underscores are added to ensure that there will be no clash with existing free symbols.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: 0_}
```

classmethod class_key()

Nice order of classes.

compare(other)

Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the “other” then their classes are ordered according to the `sorted_classes` list.

Examples

```
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

count(query)

Count the number of matching subexpressions.

count_ops(visual=None)

wrapper for `count_ops` that returns the operation count.

doit(**hints)

Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via ‘hints’ or unless the ‘deep’ hint was set to ‘False’.

```
>>> from sympy import Integral
>>> from sympy.abc import x
```

```
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(other, symbol=None)

Compare two expressions and handle dummy symbols.

Examples

```
>>> from sympy import Dummy
>>> from sympy.abc import x, y
```

```
>>> u = Dummy('u')
```

```
>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

find(query, group=False)

Find all subexpressions matching a query.

free_symbols

Return from the atoms of self those which are free symbols.

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

classmethod fromiter(args, **assumptions)

Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

Examples

```
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

func

The top-level function in an expression.

The following should hold for all objects:

```
>> x == x.func(*x.args)
```

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

has(*patterns)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note `has` is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```
>>> from sympy.sets import Interval
>>> i = Interval.Lopen(0, 5); i
Interval.Lopen(0, 5)
>>> i.args
(0, 5, True, False)
>>> i.has(4) # there is no "4" in the arguments
False
>>> i.has(0) # there is a "0" in the arguments
True
```

Instead, use `contains` to determine whether a number is in the interval or not:

```
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

`is_comparable`

Return `True` if `self` can be computed to a real number (or already is a real number) with precision, else `False`.

Examples

```
>>> from sympy import exp_polar, pi, I
>>> (I*exp_polar(I*pi/2)).is_comparable
True
>>> (I*exp_polar(I*pi*2)).is_comparable
False
```

A `False` result does not mean that *self* cannot be rewritten into a form that would be comparable. For example, the difference computed below is zero but without simplification it does not evaluate to a zero with precision:

```
>>> e = 2**pi*(1 + 2**pi)
>>> dif = e - e.expand()
```

(continues on next page)

(continued from previous page)

```
>>> dif.is_comparable
False
>>> dif.n(2)._prec
1
```

match(pattern, old=False)

Pattern matching.

Wild symbols match all.

Return None when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> from sympy import Wild
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

The old flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless old=True:

```
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(expr, repl_dict={}, old=False)

Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
```

(continues on next page)

(continued from previous page)

```
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

rcall(*args)

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance in SymPy the the following will not work:

```
(x+Lambda(y, 2*y))(z) == x+2*z,
```

however you can use

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

replace(query, value, map=False, simultaneous=True, exact=False)

Replace matching subexpressions of `self` with `value`.

If `map = True` then also return the mapping `{old: new}` where `old` was a sub-expression found with `query` and `new` is the replacement value for it. If the expression itself doesn't match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to `False`. In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is `True`, then the match will only succeed if non-zero values are received for each Wild that appears in the match pattern.

The list of possible combinations of queries and replacement values is listed below:

See also:

subs (page 107) substitution of subexpressions as defined by the objects themselves.

xreplace (page 109) exact node replacement in expr tree; also capable of using matching rules

Examples

Initial setup

```
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type obj.replace(type, newtype)

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a = Wild('a')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

When the default value of `False` is used with patterns that have more than one Wild symbol, non-intuitive results may be obtained:

```
>>> b = Wild('b')
>>> (2*x).replace(a*x + b, b - a)
2/x
```

For this reason, the `exact` option can be used to make the replacement only when the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a, exact=True)
y - 2
>>> (2*x).replace(a*x + b, b - a, exact=True)
2*x
```

2.2. pattern -> func obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func obj.replace(filter, func)

Replace subexpression `e` with `func(e)` if `filter(e)` is `True`.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

rewrite(*args, **hints)

Rewrite functions in terms of other functions.

Rewrites expression containing applications of functions of one kind in terms of functions of different kind. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

As a pattern this function accepts a list of functions to to rewrite (instances of DefinedFunction class). As rule you can use string or a destination function instance (in this case rewrite() will use the str() function).

There is also the possibility to pass hints on how to rewrite the given expressions. For now there is only one such hint defined called 'deep'. When 'deep' is set to False it will forbid functions to rewrite their contents.

Examples

```
>>> from sympy import sin, exp
>>> from sympy.abc import x
```

Unspecified pattern:

```
>>> sin(x).rewrite(exp)
-I*(exp(I*x) - exp(-I*x))/2
```

Pattern as a single function:

```
>>> sin(x).rewrite(sin, exp)
-I*(exp(I*x) - exp(-I*x))/2
```

Pattern as a list of functions:

```
>>> sin(x).rewrite([sin, ], exp)
-I*(exp(I*x) - exp(-I*x))/2
```

sort_key(order=None)

Return a sort key.

Examples

```
>>> from sympy.core import S, I
```



```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(*args, **kwargs)

Substitutes old for new in an expression after sympifying args.

args is either:

- two arguments, e.g. `foo.subs(old, new)`
- **one iterable argument, e.g. `foo.subs(iterable)`. The iterable may be**
 - o an iterable container with (old, new) pairs. In this case the** replacements are processed in the order given with successive patterns possibly affecting replacements already made.
 - o a dict or set whose key/value items correspond to old/new pairs.** In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the `default_sort_key`. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is `True`, the subexpressions will not be evaluated until all the substitutions have been made.

See also:

replace (page 104) replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

xreplace (page 109) exact node replacement in expr tree; also capable of using matching rules

evalf calculates the given formula to a desired level of precision

Examples

```
>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the `x**2` but not the `x**4`, use `xreplace`:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to `True`:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by `count_op` length, number of arguments and by the default `sort_key` to break any ties. All other iterables are left unsorted.

```
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e
```

```
>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
>>> E = (exp(x), e)
```

```
>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)
```

```
>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```
>>> (x**3 - 3*x).subs({x: oo})
nan
```

```
>>> limit(x**3 - 3*x, x, oo)
oo
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to `evalf` as

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.333333333333333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.33333333333333333333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

xreplace(rule)

Replace occurrences of objects within the expression.

Parameters rule : dict-like

Expresses a replacement rule

Returns xreplace : the result of the replacement

See also:

replace (page 104) replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs (page 107) substitution of subexpressions as defined by the objects themselves.

Examples

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

xreplace doesn't differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

Atom

class `sympy.core.basic.Atom`

A parent class for atomic things. An atom is an expression with no subexpressions.

Examples

Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

5.1.5 core

5.1.6 singleton

S

class `sympy.core.singleton.SingletonRegistry`

The registry for the singleton classes (accessible as S).

This class serves as two separate things.

The first thing it is is the `SingletonRegistry`. Several classes in SymPy appear so often that they are singletonized, that is, using some metaprogramming they are made so that they can only be instantiated once (see the `sympy.core.singleton.Singleton` class for details). For instance, every time you create `Integer(0)`, this will return the same instance, [sympy.core.numbers.Zero](#) (page 144). All singleton instances are attributes of the S object, so `Integer(0)` can also be accessed as `S.Zero`.

Singletonization offers two advantages: it saves memory, and it allows fast comparison. It saves memory because no matter how many times the singletonized objects appear in expressions in memory, they all point to the same single instance in memory. The fast comparison comes from the fact that you can use `is` to compare exact instances in Python (usually, you need to use `==` to compare things). `is` compares objects by memory address, and is very fast. For instance

```
>>> from sympy import S, Integer
>>> a = Integer(0)
>>> a is S.Zero
True
```

For the most part, the fact that certain objects are singletonized is an implementation detail that users shouldn't need to worry about. In SymPy library code, `is` comparison is often used for performance purposes. The primary advantage of S for end users is the convenient access to certain instances that are otherwise difficult to type, like `S.Half` (instead of `Rational(1, 2)`).

When using `is` comparison, make sure the argument is sympified. For instance,

```
>>> 0 is S.Zero
False
```

This problem is not an issue when using `==`, which is recommended for most use-cases:

```
>>> 0 == S.Zero
True
```

The second thing `S` is is a shortcut for `sympy.core.sympify.sympify()` (page 91). `sympy.core.sympify.sympify()` (page 91) is the function that converts Python objects such as `int(1)` into SymPy objects such as `Integer(1)`. It also converts the string form of an expression into a SymPy expression, like `sympify("x**2") -> Symbol("x")**2`. `S(1)` is the same thing as `sympify(1)` (basically, `S.__call__` has been defined to call `sympify`).

This is for convenience, since `S` is a single letter. It's mostly useful for defining rational numbers. Consider an expression like $x + 1/2$. If you enter this directly in Python, it will evaluate the $1/2$ and give `0.5` (or just `0` in Python 2, because of integer division), because both arguments are ints (see also [Two Final Notes: ^ and /](#) (page 11)). However, in SymPy, you usually want the quotient of two integers to give an exact rational number. The way Python's evaluation works, at least one side of an operator needs to be a SymPy object for the SymPy evaluation to take over. You could write this as `x + Rational(1, 2)`, but this is a lot more typing. A shorter version is `x + S(1)/2`. Since `S(1)` returns `Integer(1)`, the division will return a Rational type, since it will call `Integer.__div__`, which knows how to return a Rational.

5.1.7 expr

5.1.8 Expr

class `sympy.core.expr.Expr`

Base class for algebraic expressions.

Everything that requires arithmetic operations to be defined should subclass this class, instead of `Basic` (which should be used only for argument storage and expression manipulation, i.e. pattern matching, substitutions, etc).

See also:

`sympy.core.basic.Basic` (page 96)

apart(`x=None`, `**args`)

See the `apart` function in `sympy.polys`

args_cnc(`cset=False`, `warn=True`, `split_1=True`)

Return [commutative factors, non-commutative factors] of self.

self is treated as a `Mul` and the ordering of the factors is maintained. If `cset` is `True` the commutative factors will be returned in a set. If there were repeated factors (as may happen with an unevaluated `Mul`) then an error will be raised unless it is explicitly suppressed by setting `warn` to `False`.

Note: `-1` is always separated from a `Number` unless `split_1` is `False`.

```
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
```

(continues on next page)

(continued from previous page)

```
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]
```

The arg is always treated as a Mul:

```
>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_coeff_Add(rational=False)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

as_coeff_add(*deps)

Return the tuple (c, args) where self is written as an Add, a.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of a; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you don't know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you don't want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use self.as_independent(*deps)

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(x)

$c*x^e \rightarrow c, e$ where x can be any symbolic expression.

as_coeff_mul(*deps, **kwargs)

Return the tuple (c, args) where self is written as a Mul, m.

c should be a Rational multiplied by any factors of the Mul that are independent of deps.

args should be a tuple of all other factors of m; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you don't know if self is a Mul or not but you want to treat self as a Mul or if you want to process the individual arguments of the tail of self as a Mul.

- if you know self is a Mul and want only the head, use self.args[0];
- if you don't want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail;
- if you want to split self into an independent and dependent parts use self.as_independent(*deps)

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())
```

as_coefficient(expr)

Extracts symbolic coefficient at the given expression. In other words, this functions separates 'self' into the product of 'expr' and 'expr'-free coefficient. If such separation is not possible it will return None.

See also:

coeff (page 119) return sum of terms have a given factor

as_coeff_Add (page 112) separate the additive constant from an expression

as_coeff_Mul (page 112) separate the multiplicative constant from an expression

as_independent (page 115) separate x-dependent terms/factors from others

sympy.polys.polytools.coeff_monomial efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.nth like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
```

(continues on next page)

(continued from previous page)

```
>>> _args[0] # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, `None` is returned. (If the coefficient $2*x$ is desired then the `coeff` method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

as_coefficients_dict()

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```
>>> from sympy.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

as_content_primitive(radical=False, clear=True)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. `expand_mul` should not be applied to self).

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import x, y, z

>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```


The `as_content_primitive` function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their `as_content_primitives` are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((5*(x*(1 + y)) + 2.0*x*(3 + 3*y))**2).as_content_primitive()
(1, 121.0*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If `clear=False` (default is `True`) then content will not be removed from an `Add` if it can be distributed to leave one or more terms with integer coefficients.

```
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

`as_expr(*gens)`

Convert a polynomial to a SymPy expression.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y
```

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

`as_independent(*deps, **hint)`

A mostly naive separation of a `Mul` or `Add` into arguments that are not are dependent

on deps. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- `separatevars()` to change Mul, Add and Pow (including exp) into Mul
- `.expand(mul=True)` to change Add or Mul into Add
- `.expand(log=True)` to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for *self* of zero regardless of hints.

For nonzero *self*, the returned tuple (i, d) has the following interpretation:

- i will have no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when *self* is an Add) or 1 (when *self* is a Mul)
- if *self* is an Add then $\text{self} = i + d$
- if *self* is a Mul then $\text{self} = i*d$
- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint `as_Add=True`

See also:

[separatevars](#) (page 1157), [expand](#) (page 122), [Add.as_two_terms](#), [Mul.as_two_terms](#), [as_coeff_add](#) (page 112), [as_coeff_mul](#) (page 112)

Examples

- self is an Add

```
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z

>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

- self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when self is a Mul

```
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
```

(continues on next page)

(continued from previous page)

```
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

- self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

- force self to be treated as an Add:

```
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

- force self to be treated as a Mul:

```
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

- use **.as_independent()** for true independence testing instead of **.has()**. The former considers only symbols in the free symbols while the latter considers all symbols

```
>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
```

(continues on next page)

(continued from previous page)

```

>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))

```

as_leading_term(*symbols)

Returns the leading (nonzero) term of the series expansion of self.

The `_eval_as_leading_term` routines are used to do this, and they must always return a non-zero value.

Examples

```

>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)

```

as_numer_denom()

expression -> a/b -> a, b

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

normal return a/b instead of a, b

as_ordered_factors(order=None)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(order=None, data=False)

Transform an expression to an ordered list of terms.

Examples

```

>>> from sympy import sin, cos
>>> from sympy.abc import x

```

```

>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]

```

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and

contains non-commutative factors since the order that they appeared will be lost in the dictionary.

as_real_imag(deep=True, **hints)

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method can't be confused with `re()` and `im()` functions, which does not perform complex expansion at evaluation.

However it is possible to expand both `re()` and `im()` functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_terms()

Transform an expression to a list of terms.

cancel(*gens, **args)

See the `cancel` function in `sympy.polys`

coeff(x, n=1, right=False)

Returns the coefficient from the term(s) containing x^n . If n is zero then all terms independent of x will be returned.

When x is noncommutative, the coefficient to the left (default) or right of x can be returned. The keyword 'right' is ignored when x is commutative.

See also:

as_coefficient (page 113) separate the expression into a coefficient and factor

as_coeff_Add (page 112) separate the additive constant from an expression

as_coeff_Mul (page 112) separate the multiplicative constant from an expression

as_independent (page 115) separate x -dependent terms/factors from others

sympy.polys.polytools.coeff_monomial efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.nth like `coeff_monomial` but powers of monomial terms are used

Examples

```
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making $n=0$; in this case `expr.as_independent(x)[0]` is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so $1 + z*(1 + y)$ is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, `factor_terms` can be used first:

```
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

collect(syms, func=None, evaluate=True, exact=False, distribute_order_term=True)

See the collect function in `sympy.simplify`

combsimp()

See the combsimp function in `sympy.simplify`

compute_leading_term(x, logx=None)

as_leading_term is only allowed for results of `.series()` This is a wrapper to compute a series first.

could_extract_minus_sign()

Return True if self is not in a canonical form with respect to its sign.

For most expressions, e, there will be a difference in e and -e. When there is, True will be returned for one and False for the other; False will be returned if there is no difference.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

count_ops(visual=None)

wrapper for count_ops that returns the operation count.

equals(other, failing_expression=False)

Return True if self == other, False if it doesn't, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.

If self is a Number (or complex number) that is not zero, then the result is False.

If self is a number and has not evaluated to zero, evalf will be used to test whether the expression evaluates to zero. If it does so and the result has significance (i.e.

the precision is either -1, for a Rational result, or is greater than 1) then the evalf value will be used to return True or False.

expand(deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints)
Expand an expression using hints.

See the docstring of the `expand()` function in `sympy.core.function` for more information.

expr_free_symbols

Like `free_symbols`, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, don't return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(c)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

See also:

[extract_multiplicatively](#) (page 123), [coeff](#) (page 119), [as_coefficient](#) (page 113)

Examples

```
>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

Sometimes auto-expansion will return a less simplified result than desired; `gcd_terms` might be used in such cases:


```
>>> from sympy import gcd_terms
>>> (4*x*(y + 1) + y).extract_additively(x)
4*x*(y + 1) + x*(4*y + 3) - x*(4*y + 4) + y
>>> gcd_terms(_)
x*(4*y + 3) + y
```

extract_branch_factor(allow_half=False)

Try to write self as $\exp_{\text{polar}}(2\pi i n)z$ in a nice way. Return (z, n).

```
>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If allow_half is True, also extract $\exp_{\text{polar}}(I\pi)$:

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(c)

Return None if it's not possible to make self in the form $c * \text{something}$ in a nice way, i.e. preserving the properties of arguments of self.

```
>>> from sympy import symbols, Rational
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2
```

```
>>> ((x*y)**3).extract_multiplicatively(x**4 * y)
```

```
>>> (2*x).extract_multiplicatively(2)
x
```

```
>>> (2*x).extract_multiplicatively(3)
```

```
>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(*gens, **args)

See the `factor()` function in `sympy.polys.polytools`

fourier_series(limits=None)

Compute fourier sine/cosine series of self.

See the docstring of the `fourier_series()` (page 124) in `sympy.series.fourier` for more information.

fps(x=None, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)

Compute formal power power series of self.

See the docstring of the `fps()` (page 124) function in `sympy.series.formal` for more information.

gammasimp()

See the `gammasimp` function in `sympy.simplify`

get0()

Returns the additive $O(\cdot)$ symbol if there is one, else None.

getn()

Returns the order of the expression.

The order is determined either from the $O(\dots)$ term. If there is no $O(\dots)$ term, it returns None.

Examples

```
>>> from sympy import 0
>>> from sympy.abc import x
>>> (1 + x + 0(x**2)).getn()
2
>>> (1 + x).getn()
```

integrate(*args, **kwargs)

See the `integrate` function in `sympy.integrals`

invert(g, *gens, **args)

Return the multiplicative inverse of `self` mod `g` where `self` (and `g`) may be symbolic expressions).

See also:

`sympy.core.numbers.mod_inverse`, `sympy.polys.polytools.invert` (page 807)

is_algebraic_expr(*syms)

This tests whether a given expression is algebraic or not, in the given symbols, `syms`. When `syms` is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “algebraic expressions” with symbolic exponents. This is a simple extension to the `is_rational_function`, including rational exponentiation.

See also:

`is_rational_function` (page 128)

References

- http://en.wikipedia.org/wiki/Algebraic_expression

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

`is_constant(*wrt, **flags)`

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

If an expression has no free symbols then it is a constant. If there are free symbols it is possible that the expression is a constant, perhaps (but not necessarily) zero. To test such expressions, two strategies are tried:

1) numerical evaluation at two random points. If two such evaluations give two different values and the values have a precision greater than 1 then self is not constant. If the evaluations agree or could not be obtained with any precision, no decision is made. The numerical testing is done only if wrt is different than the free symbols.

2) differentiation with respect to variables in 'wrt' (or all free symbols if omitted) to see if the expression is constant or not. This will not always lead to an expression that is zero even though an expression is constant (see added test in test_expr.py). If all derivatives are zero then self is constant with respect to the given symbols.

If neither evaluation nor differentiation can prove the expression is constant, None is returned unless two numerical values happened to be the same and the flag `fail-ing_number` is True - in that case the numerical value will be returned.

If flag `simplify=False` is passed, self will not be simplified; the default is True since self should be simplified before testing.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
```

(continues on next page)

(continued from previous page)

```

True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True

```

```

>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0 or 1
True

```

is_number

Returns True if self has no free symbols and no undefined functions (AppliedUndef, to be precise). It will be faster than if not self.free_symbols, however, since is_number will fail as soon as it hits a free symbol or undefined function.

See also:

sympy.core.basic.is_comparable

Examples

```

>>> from sympy import log, Integral, cos, sin, pi
>>> from sympy.core.function import Function
>>> from sympy.abc import x
>>> f = Function('f')

```

```

>>> x.is_number
False
>>> f(1).is_number
False
>>> (2*x).is_number
False
>>> (2 + Integral(2, x)).is_number
False
>>> (2 + Integral(2, (x, 1, 2))).is_number
True

```

Not all numbers are Numbers in the SymPy sense:

```
>>> pi.is_number, pi.is_Number
(True, False)
```

If something is a number it should evaluate to a number with real and imaginary parts that are Numbers; the result may not be comparable, however, since the real and/or imaginary part of the result may not have precision.

```
>>> cos(1).is_number and cos(1).is_comparable
True
```

```
>>> z = cos(1)**2 + sin(1)**2 - 1
>>> z.is_number
True
>>> z.is_comparable
False
```

is_polynomial(*syms)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are “polynomials” with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and Poly(expr, *syms) should work if and only if expr.is_polynomial(*syms) returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do Symbol('z', polynomial=True).

Examples

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
```

(continues on next page)

(continued from previous page)

```
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also `.is_rational_function()`

is_rational_function(*syms)

Test whether function is a ratio of two polynomials in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “rational functions” with symbolic exponents. Thus, you should be able to call `.as_numer_denom()` and apply polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do `Symbol('z', rational_function=True)`.

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.abc import x, y
```

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
```

(continues on next page)

(continued from previous page)

```
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also `is_algebraic_expr()`.

leadterm(x)

Returns the leading term $a*x^b$ as a tuple (a, b).

Examples

```
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

limit(x, xlim, dir='+')

Compute limit $x \rightarrow xlim$.

lseries(x=None, x0=0, dir='+', logx=None)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for example, will never terminate. It will just keep printing terms of the $\sin(x)$ series:

```
for term in sin(x).lseries():
    print term
```

The advantage of `lseries()` over `nseries()` is that many times you are just interested in the next term in the series (i.e. the first term for example), but you don't know how many you should ask for in `nseries()` using the "n" parameter.

See also `nseries()`.

nseries(x=None, x0=0, n=6, dir='+', logx=None)

Wrapper to `_eval_nseries` if assumptions allow, else to `series`.

If x is given, x_0 is 0, $dir='+'$, and self has x , then `_eval_nseries` is called. This calculates "n" terms in the innermost expressions and then builds up the final series just by "cross-multiplying" everything out.

The optional `logx` parameter can be used to replace any $\log(x)$ in the returned series with a symbolic value to avoid evaluating $\log(x)$ at 0. A symbol to use in place of $\log(x)$ should be provided.

Advantage - it's fast, because we don't have to determine how many terms we need to calculate in advance.

Disadvantage - you may end up with less terms than you may have expected, but the $O(x^n)$ term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to `series` which will try harder to return the correct number of terms.

See also `lseries()`.

Examples

```
>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the `logx` parameter — in the following example the expansion fails since `sin` does not have an asymptotic expansion at $-\infty$ (the limit of $\log(x)$ as x approaches 0):

```
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but gives only an Order term unless the `logx` parameter is used:

```
>>> e = x**y
>>> e.nseries(x, 0, 2)
O(log(x)**2)
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(constants=[], tolerance=None, full=False)

See the `nsimplify` function in `sympy.simplify`

powsimp(*args, **kwargs)

See the `powsimp` function in `sympy.simplify`

primitive()

Return the positive Rational that can be extracted non-recursively from every term of self (i.e., self is treated like an Add). This is like the `as_coeff_Mul()` method but `primitive` always extracts a positive Rational (never a negative or a Float).

Examples

```
>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```


radsimp(**kwargs)

See the radsimp function in sympy.simplify

ratsimp()

See the ratsimp function in sympy.simplify

refine(assumption=True)

See the refine function in sympy.assumptions

remove0()

Removes the additive $O(\dots)$ symbol if there is one

round(p=0)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Notes

Do not confuse the Python builtin function, round, with the SymPy method of the same name. The former always returns a float (or raises an error if applied to a complex value) while the latter returns either a Number or a complex number:

```
>>> isinstance(round(S(123), -2), Number)
False
>>> isinstance(S(123).round(-2), Number)
True
>>> isinstance((3*I).round(), Mul)
True
>>> isinstance((1 + 3*I).round(), Add)
True
```

Examples

```
>>> from sympy import pi, E, I, S, Add, Mul, Number
>>> S(10.5).round()
11.
>>> pi.round()
3.
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6. + 3.*I
```

The round method has a chopping effect:

```
>>> (2*pi + I/10).round()
6.
>>> (pi/10 + 2*I).round()
2.*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

separate(deep=False, force=False)

See the separate function in sympy.simplify

series(x=None, x0=0, n=6, dir='+', logx=None)

Series expansion of “self” around $x = x_0$ yielding either terms of the series one by one (the lazy series given when $n=None$), else all the terms at once when $n \neq None$.

Returns the series expansion of “self” around the point $x = x_0$ with respect to x up to $O((x - x_0)^{n+1})$ (default n is 6).

If $x=None$ and `self` is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

```
>>> from sympy import cos, exp
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)
```

If $n=None$ then a generator of the series terms will be returned.

```
>>> term=cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For $dir=+$ (default) the series is calculated from the right and for $dir=-$ the series from the left. For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
```

simplify(ratio=1.7, measure=None)

See the `simplify` function in `sympy.simplify`

taylor_term(n, x, *previous_terms)

General method for the Taylor term.

This method is slow, because it differentiates n -times. Subclasses can redefine it to make it faster by using the “previous_terms”.

together(*args, **kwargs)

See the `together` function in `sympy.polys`

trigsimp(*args)

See the `trigsimp` function in `sympy.simplify`

5.1.9 UnevaluatedExpr

class `sympy.core.expr.UnevaluatedExpr`

Expression that is not evaluated unless released.

Examples

```
>>> from sympy import UnevaluatedExpr
>>> from sympy.abc import a, b, x, y
>>> x*(1/x)
1
>>> x*UnevaluatedExpr(1/x)
x*1/x
```

5.1.10 AtomicExpr

class sympy.core.expr.AtomicExpr

A parent class for object which are both atoms and Exprs.

For example: Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

5.1.11 symbol

Symbol

class sympy.core.symbol.Symbol

Assumptions: commutative = True

You can override the default assumptions in the constructor:

```
>>> from sympy import symbols
>>> A,B = symbols('A,B', commutative = False)
>>> bool(A*B != B*A)
True
>>> bool(A*B*2 == 2*A*B) == True # multiplication by scalars is commutative
True
```

as_dummy()

Return a Dummy having the same name and same assumptions as self.

Wild

class sympy.core.symbol.Wild

A Wild symbol matches anything, or anything without whatever is explicitly excluded.

Parameters **name** : str

Name of the Wild instance.

exclude : iterable, optional

Instances in exclude will not be matched.

properties : iterable of functions, optional

Functions, each taking an expressions as input and returns a bool. All functions in properties need to return True in order for the Wild instance to match the expression.

Examples

```
>>> from sympy import Wild, WildFunction, cos, pi
>>> from sympy.abc import x, y, z
>>> a = Wild('a')
>>> x.match(a)
{a_: x}
>>> pi.match(a)
{a_: pi}
>>> (3*x**2).match(a*x)
{a_: 3*x}
>>> cos(x).match(a)
{a_: cos(x)}
>>> b = Wild('b', exclude=[x])
>>> (3*x**2).match(b*x)
>>> b.match(a)
{a_: b_}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A_}
```

Tips

When using Wild, be sure to use the exclude keyword to make the pattern more precise. Without the exclude pattern, you may get matches that are technically correct, but not what you wanted. For example, using the above without exclude:

```
>>> from sympy import symbols
>>> a, b = symbols('a b', cls=Wild)
>>> (2 + 3*y).match(a*x + b*y)
{a_: 2/x, b_: 3}
```

This is technically correct, because $(2/x)*x + 3*y == 2 + 3*y$, but you probably wanted it to not match at all. The issue is that you really didn't want a and b to include x and y, and the exclude parameter lets you specify exactly this. With the exclude parameter, the pattern will not match.

```
>>> a = Wild('a', exclude=[x, y])
>>> b = Wild('b', exclude=[x, y])
>>> (2 + 3*y).match(a*x + b*y)
```

Exclude also helps remove ambiguity from matches.

```
>>> E = 2*x**3*y*z
>>> a, b = symbols('a b', cls=Wild)
>>> E.match(a*b)
{a_: 2*y*z, b_: x**3}
>>> a = Wild('a', exclude=[x, y])
>>> E.match(a*b)
{a_: z, b_: 2*x**3*y}
>>> a = Wild('a', exclude=[x, y, z])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Wild also accepts a properties parameter:

```
>>> a = Wild('a', properties=[lambda k: k.is_Integer])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Dummy

class sympy.core.symbol.Dummy

Dummy symbols are each unique, even if they have the same name:

```
>>> from sympy import Dummy
>>> Dummy("x") == Dummy("x")
False
```

If a name is not supplied then a string value of an internal count will be used. This is useful when a temporary variable is needed and the name of the variable used in the expression is not important.

```
>>> Dummy()
_Dummy_10
```

symbols

sympy.core.symbol.symbols(names, **args)

Transform strings into instances of [Symbol](#) (page 133) class.

[symbols\(\)](#) (page 135) function returns a sequence of symbols with names taken from names argument, which can be a comma or whitespace delimited string, or a sequence of strings:

```
>>> from sympy import symbols, Function

>>> x, y, z = symbols('x,y,z')
>>> a, b, c = symbols('a b c')
```

The type of output is dependent on the properties of input arguments:

```
>>> symbols('x')
x
>>> symbols('x,')
(x,)
>>> symbols('x,y')
(x, y)
>>> symbols(('a', 'b', 'c'))
(a, b, c)
>>> symbols(['a', 'b', 'c'])
[a, b, c]
>>> symbols({'a', 'b', 'c'})
{a, b, c}
```

If an iterable container is needed for a single symbol, set the seq argument to True or terminate the symbol name with a comma:

```
>>> symbols('x', seq=True)
(x,)
```

To reduce typing, range syntax is supported to create indexed symbols. Ranges are indicated by a colon and the type of range is determined by the character to the right of the colon. If the character is a digit then all contiguous digits to the left are taken as the nonnegative starting value (or 0 if there is no digit left of the colon) and all contiguous digits to the right are taken as 1 greater than the ending value:

```
>>> symbols('x:10')
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

>>> symbols('x5:10')
(x5, x6, x7, x8, x9)
>>> symbols('x5(:2)')
(x50, x51)

>>> symbols('x5:10,y:5')
(x5, x6, x7, x8, x9, y0, y1, y2, y3, y4)

>>> symbols(('x5:10', 'y:5'))
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))
```

If the character to the right of the colon is a letter, then the single letter to the left (or 'a' if there is none) is taken as the start and all characters in the lexicographic range through the letter to the right are used as the range:

```
>>> symbols('x:z')
(x, y, z)
>>> symbols('x:c') # null range
()
>>> symbols('x(:c)')
(xa, xb, xc)

>>> symbols(':c')
(a, b, c)

>>> symbols('a:d, x:z')
(a, b, c, d, x, y, z)

>>> symbols(('a:d', 'x:z'))
((a, b, c, d), (x, y, z))
```

Multiple ranges are supported; contiguous numerical ranges should be separated by parentheses to disambiguate the ending number of one range from the starting number of the next:

```
>>> symbols('x:2(1:3)')
(x01, x02, x11, x12)
>>> symbols(':3:2') # parsing is from left to right
(00, 01, 10, 11, 20, 21)
```

Only one pair of parentheses surrounding ranges are removed, so to include parentheses around ranges, double them. And to include spaces, commas, or colons, escape them with a backslash:

```
>>> symbols('x((a:b))')
(x(a), x(b))
>>> symbols(r'x(:1\,:2)') # or r'x((:1)\,(:2))'
(x(0,0), x(0,1))
```

All newly created symbols have assumptions set according to args:

```
>>> a = symbols('a', integer=True)
>>> a.is_integer
True

>>> x, y, z = symbols('x,y,z', real=True)
>>> x.is_real and y.is_real and z.is_real
True
```

Despite its name, `symbols()` (page 135) can create symbol-like objects like instances of Function or Wild classes. To achieve this, set `cls` keyword argument to the desired type:

```
>>> symbols('f,g,h', cls=Function)
(f, g, h)

>>> type(_[0])
<class 'sympy.core.function.UndefinedFunction'>
```

var

`sympy.core.symbol.var(names, **args)`

Create symbols and inject them into the global namespace.

This calls `symbols()` (page 135) with the same arguments and puts the results into the global namespace. It's recommended not to use `var()` (page 137) in library code, where `symbols()` (page 135) has to be used:

```
.. rubric:: Examples
```

```
>>> from sympy import var
```

```
>>> var('x')
x
>>> x
x
```

```
>>> var('a,ab,abc')
(a, ab, abc)
>>> abc
abc
```

```
>>> var('x,y', real=True)
(x, y)
>>> x.is_real and y.is_real
True
```

See `symbol()` documentation for more details on what kinds of arguments can be passed to `var()` (page 137).

5.1.12 numbers

Number

class `sympy.core.numbers.Number`

Represents atomic numbers in SymPy.

Floating point numbers are represented by the `Float` class. Rational numbers (of any size) are represented by the `Rational` class. Integer numbers (of any size) are represented by the `Integer` class. `Float` and `Rational` are subclasses of `Number`; `Integer` is a subclass of `Rational`.

For example, $2/3$ is represented as `Rational(2, 3)` which is a different object from the floating point number obtained with Python division $2/3$. Even for numbers that are exactly represented in binary, there is a difference between how two forms, such as `Rational(1, 2)` and `Float(0.5)`, are used in SymPy. The rational form is to be preferred in symbolic computations.

Other kinds of numbers, such as algebraic numbers `sqrt(2)` or complex numbers $3 + 4*I$, are not instances of `Number` class as they are not atomic.

See also:

[Float](#) (page 138), [Integer](#) (page 143), [Rational](#) (page 141)

as_coeff_Add(rational=False)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

cofactors(other)

Compute GCD and cofactors of *self* and *other*.

gcd(other)

Compute GCD of *self* and *other*.

lcm(other)

Compute LCM of *self* and *other*.

Float

class `sympy.core.numbers.Float`

Represent a floating-point number of arbitrary precision.

Notes

Floats are inexact by their nature unless their value is a binary-exact value.

```
>>> approx, exact = Float(.1, 1), Float(.125, 1)
```

For calculation purposes, `evalf` needs to be able to change the precision but this will not increase the accuracy of the inexact value. The following is the most accurate 5-digit approximation of a value of 0.1 that had only 1 digit of precision:

```
>>> approx.evalf(5)
0.099609
```


By contrast, 0.125 is exact in binary (as it is in base 10) and so it can be passed to Float or evalf to obtain an arbitrary precision with matching accuracy:

```
>>> Float(exact, 5)
0.12500
>>> exact.evalf(20)
0.12500000000000000000
```

Trying to make a high-precision Float from a float is not disallowed, but one must keep in mind that the underlying float (not the apparent decimal value) is being obtained with high precision. For example, 0.3 does not have a finite binary representation. The closest rational is the fraction $5404319552844595/2^{54}$. So if you try to obtain a Float of 0.3 to 20 digits of precision you will not see the same thing as 0.3 followed by 19 zeros:

```
>>> Float(0.3, 20)
0.29999999999999998890
```

If you want a 20-digit value of the decimal 0.3 (not the floating point approximation of 0.3) you should send the 0.3 as a string. The underlying representation is still binary but a higher precision than Python's float is used:

```
>>> Float('0.3', 20)
0.30000000000000000000
```

Although you can increase the precision of an existing Float using Float it will not increase the accuracy - the underlying value is not changed:

```
>>> def show(f): # binary rep of Float
...     from sympy import Mul, Pow
...     s, m, e, b = f._mpf_
...     v = Mul(int(m), Pow(2, int(e), evaluate=False), evaluate=False)
...     print('%s at prec=%s' % (v, f._prec))
...
>>> t = Float('0.3', 3)
>>> show(t)
4915/2**14 at prec=13
>>> show(Float(t, 20)) # higher prec, not higher accuracy
4915/2**14 at prec=70
>>> show(Float(t, 2)) # lower prec
307/2**10 at prec=10
```

The same thing happens when evalf is used on a Float:

```
>>> show(t.evalf(20))
4915/2**14 at prec=70
>>> show(t.evalf(2))
307/2**10 at prec=10
```

Finally, Floats can be instantiated with an mpf tuple (n, c, p) to produce the number $(-1)^n c 2^p$:

```
>>> n, c, p = 1, 5, 0
>>> (-1)**n*c*2**p
-5
>>> Float((1, 5, 0))
-5.000000000000000
```

An actual mpf tuple also contains the number of bits in c as the last element of the tuple:

```
>>> _._mpf_
(1, 5, 0, 3)
```

This is not needed for instantiation and is not the same thing as the precision. The mpf tuple and the precision are two separate quantities that Float tracks.

Examples

```
>>> from sympy import Float
>>> Float(3.5)
3.500000000000000
>>> Float(3)
3.000000000000000
```

Creating Floats from strings (and Python int and long types) will give a minimum precision of 15 digits, but the precision will automatically increase to capture all digits entered.

```
>>> Float(1)
1.000000000000000
>>> Float(10**20)
100000000000000000000.
>>> Float('1e20')
100000000000000000000.
```

However, floating-point numbers (Python float types) retain only 15 digits of precision:

```
>>> Float(1e20)
1.000000000000000e+20
>>> Float(1.23456789123456789)
1.23456789123457
```

It may be preferable to enter high-precision decimal numbers as strings:

```
Float('1.23456789123456789') 1.23456789123456789
```

The desired number of digits can also be specified:

```
>>> Float('1e-3', 3)
0.00100
>>> Float(100, 4)
100.0
```

Float can automatically count significant figures if a null string is sent for the precision; space are also allowed in the string. (Auto-counting is only allowed for strings, ints and longs).

```
>>> Float('123 456 789 . 123 456', '')
123456789.123456
>>> Float('12e-3', '')
0.012
>>> Float(3, '')
3.
```

If a number is written in scientific notation, only the digits before the exponent are considered significant if a decimal appears, otherwise the “e” signifies only how to move the decimal:

```
>>> Float('60.e2', '') # 2 digits significant
6.0e+3
>>> Float('60e2', '') # 4 digits significant
6000.
>>> Float('600e-2', '') # 3 digits significant
6.00
```

Attributes

is_irrational	
is_rational	

Rational

class `sympy.core.numbers.Rational`

Represents rational numbers (p/q) of any size.

See also:

`sympify`, `sympy.simplify.simplify.nsimpify` (page 1159)

Examples

```
>>> from sympy import Rational, nsimplify, S, pi
>>> Rational(1, 2)
1/2
```

`Rational` is unprejudiced in accepting input. If a float is passed, the underlying value of the binary representation will be returned:

```
>>> Rational(.5)
1/2
>>> Rational(.2)
3602879701896397/18014398509481984
```

If the simpler representation of the float is desired then consider limiting the denominator to the desired value or convert the float to a string (which is roughly equivalent to limiting the denominator to 10^{**12}):

```
>>> Rational(str(.2))
1/5
>>> Rational(.2).limit_denominator(10**12)
1/5
```

An arbitrarily precise `Rational` is obtained when a string literal is passed:

```
>>> Rational("1.23")
123/100
>>> Rational('1e-2')
1/100
>>> Rational(".1")
```

(continues on next page)

(continued from previous page)

```
1/10
>>> Rational('1e-2/3.2')
1/320
```

The conversion of other types of strings can be handled by the `sympify()` function, and conversion of floats to expressions or simple fractions can be handled with `nsimplify`:

```
>>> S('.[3]') # repeating digits in brackets
1/3
>>> S('3**2/10') # general expressions
9/10
>>> nsimplify(.3) # numbers that have a simple form
3/10
```

But if the input does not reduce to a literal Rational, an error will be raised:

```
>>> Rational(pi)
Traceback (most recent call last):
...
TypeError: invalid input: pi
```

Low-level

Access numerator and denominator as `.p` and `.q`:

```
>>> r = Rational(3, 4)
>>> r
3/4
>>> r.p
3
>>> r.q
4
```

Note that `p` and `q` return integers (not SymPy Integers) so some care is needed when using them in expressions:

```
>>> r.p/r.q
0.75
```

as_coeff_Add(rational=False)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

as_content_primitive(radical=False, clear=True)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import S
>>> (S(-3)/2).as_content_primitive()
(3/2, -1)
```

See docstring of `Expr.as_content_primitive` for more examples.

factors(limit=None, use_trial=True, use_rho=False, use_pm1=False, verbose=False, visual=False)

A wrapper to `factorint` which return factors of self that are smaller than limit (or cheap to compute). Special methods of factoring are disabled by default so that only trial division is used.

limit_denominator(max_denominator=1000000)

Closest Rational to self with denominator at most max_denominator.

```
>>> from sympy import Rational
>>> Rational('3.141592653589793').limit_denominator(10)
22/7
>>> Rational('3.141592653589793').limit_denominator(100)
311/99
```

Integer

class `sympy.core.numbers.Integer`

Represents integer numbers of any size.

Examples

```
>>> from sympy import Integer
>>> Integer(3)
3
```

If a float or a rational is passed to `Integer`, the fractional part will be discarded; the effect is of rounding toward zero.

```
>>> Integer(3.8)
3
>>> Integer(-3.8)
-3
```

A string is acceptable input if it can be parsed as an integer:

```
>>> Integer("9" * 20)
99999999999999999999
```

It is rarely needed to explicitly instantiate an `Integer`, because Python integers are automatically converted to `Integer` when they are used in SymPy expressions.

NumberSymbol

class `sympy.core.numbers.NumberSymbol`

approximation(number_cls)

Return an interval with `number_cls` endpoints that contains the value of `NumberSymbol`. If not implemented, then return `None`.

RealNumber

`sympy.core.numbers.RealNumber`

alias of `sympy.core.numbers.Float` (page 138)

igcd

`sympy.core.numbers.igcd(*args)`

Computes nonnegative integer greatest common divisor.

The algorithm is based on the well known Euclid's algorithm. To improve speed, `igcd()` has its own caching mechanism implemented.

Examples

```
>>> from sympy.core.numbers import igcd
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

ilcm

`sympy.core.numbers.ilcm(*args)`

Computes integer least common multiple.

Examples

```
>>> from sympy.core.numbers import ilcm
>>> ilcm(5, 10)
10
>>> ilcm(7, 3)
21
>>> ilcm(5, 10, 15)
30
```

seterr

`sympy.core.numbers.seterr(divide=False)`

Should sympy raise an exception on 0/0 or return a nan?

divide == True raise an exception divide == False ... return nan

Zero

`class sympy.core.numbers.Zero`

The number zero.

Zero is a singleton, and can be accessed by `S.Zero`

References

[R58] (page 1853)

Examples

```
>>> from sympy import S, Integer, zoo
>>> Integer(0) is S.Zero
True
>>> 1/S.Zero
zoo
```

as_coeff_Mul(rational=False)
Efficiently extract the coefficient of a summation.

One

class sympy.core.numbers.**One**

The number one.

One is a singleton, and can be accessed by `S.One`.

References

[R59] (page 1853)

Examples

```
>>> from sympy import S, Integer
>>> Integer(1) is S.One
True
```

NegativeOne

class sympy.core.numbers.**NegativeOne**

The number negative one.

NegativeOne is a singleton, and can be accessed by `S.NegativeOne`.

See also:

[One](#) (page 145)

References

[R60] (page 1853)

Examples

```
>>> from sympy import S, Integer
>>> Integer(-1) is S.NegativeOne
True
```

Half

class sympy.core.numbers.Half

The rational number 1/2.

Half is a singleton, and can be accessed by S.Half.

References

[R61] (page 1853)

Examples

```
>>> from sympy import S, Rational
>>> Rational(1, 2) is S.Half
True
```

NaN

class sympy.core.numbers.NaN

Not a Number.

This serves as a place holder for numeric values that are indeterminate. Most operations on NaN, produce another NaN. Most indeterminate forms, such as $0/0$ or $\infty - \infty$ produce NaN. Two exceptions are 0^{**0} and ∞^{**0} , which all produce 1 (this is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`. Differences are noted below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the initially counter-intuitive results with `Eq` and `==` in the examples below.

NaN is not comparable so inequalities raise a `TypeError`. This is in contrast with floating point nan where all inequalities are false.

NaN is a singleton, and can be accessed by `S.NaN`, or can be imported as `nan`.

References

[R62] (page 1853)

Examples

```
>>> from sympy import nan, S, oo, Eq
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan)    # mathematical equality
False
>>> nan == nan      # structural equality
True
```

Attributes

is_algebraic	
is_finite	
is_integer	
is_negative	
is_positive	
is_prime	
is_rational	
is_real	
is_transcendental	
is_zero	

Infinity

class sympy.core.numbers.Infinity

Positive infinite quantity.

In real analysis the symbol ∞ denotes an unbounded limit: $x \rightarrow \infty$ means that x grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled $+\infty$ and $-\infty$ can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by `S.Infinity`, or can be imported as `oo`.

See also:

[NegativeInfinity](#) (page 148), [NaN](#) (page 146)

References

[R63] (page 1853)

Examples

```
>>> from sympy import oo, exp, limit, Symbol
>>> 1 + oo
oo
>>> 42/oo
0
>>> x = Symbol('x')
>>> limit(exp(x), x, oo)
oo
```

NegativeInfinity

class sympy.core.numbers.NegativeInfinity

Negative infinite quantity.

NegativeInfinity is a singleton, and can be accessed by `S.NegativeInfinity`.

See also:

[Infinity](#) (page 147)

ComplexInfinity

class sympy.core.numbers.ComplexInfinity

Complex infinity.

In complex analysis the symbol ∞ , called “complex infinity”, represents a quantity with infinite magnitude, but undetermined complex phase.

ComplexInfinity is a singleton, and can be accessed by `S.ComplexInfinity`, or can be imported as `zoo`.

See also:

[Infinity](#) (page 147)

Examples

```
>>> from sympy import zoo, oo
>>> zoo + 42
zoo
>>> 42/zoo
0
>>> zoo + zoo
nan
>>> zoo*zoo
zoo
```

Exp1

class sympy.core.numbers.Exp1

The e constant.

The transcendental number $e = 2.718281828\dots$ is the base of the natural logarithm and of the exponential function, $e = \exp(1)$. Sometimes called Euler's number or Napier's constant.

Exp1 is a singleton, and can be accessed by `S.Exp1`, or can be imported as `E`.

References

[R64] (page 1853)

Examples

```
>>> from sympy import exp, log, E
>>> E is exp(1)
True
>>> log(E)
1
```

ImaginaryUnit

class `sympy.core.numbers.ImaginaryUnit`

The imaginary unit, $i = \sqrt{-1}$.

I is a singleton, and can be accessed by `S.I`, or can be imported as `I`.

References

[R65] (page 1853)

Examples

```
>>> from sympy import I, sqrt
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```

Pi

class `sympy.core.numbers.Pi`

The π constant.

The transcendental number $\pi = 3.141592654\dots$ represents the ratio of a circle's circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

Pi is a singleton, and can be accessed by `S.Pi`, or can be imported as `pi`.

References

[R66] (page 1853)

Examples

```
>>> from sympy import S, pi, oo, sin, exp, integrate, Symbol
>>> S.Pi
pi
>>> pi > 3
True
>>> pi.is_irrational
True
>>> x = Symbol('x')
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)
```

EulerGamma

class sympy.core.numbers.EulerGamma

The Euler-Mascheroni constant.

$\gamma = 0.5772157\dots$ (also called Euler's constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right)$$

EulerGamma is a singleton, and can be accessed by `S.EulerGamma`.

References

[R67] (page 1853)

Examples

```
>>> from sympy import S
>>> S.EulerGamma.is_irrational
>>> S.EulerGamma > 0
True
>>> S.EulerGamma > 1
False
```

Attributes

is_irrational	

Catalan

class sympy.core.numbers.Catalan

Catalan's constant.

$K = 0.91596559\dots$ is given by the infinite series

$$K = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

Catalan is a singleton, and can be accessed by `S.Catalan`.

References

[R68] (page 1853)

Examples

```
>>> from sympy import S
>>> S.Catalan.is_irrational
>>> S.Catalan > 0
True
>>> S.Catalan > 1
False
```

Attributes

is_irrational	

GoldenRatio

class sympy.core.numbers.GoldenRatio

The golden ratio, ϕ .

$\phi = \frac{1+\sqrt{5}}{2}$ is algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum.

GoldenRatio is a singleton, and can be accessed by `S.GoldenRatio`.

References

[R69] (page 1853)

Examples

```
>>> from sympy import S
>>> S.GoldenRatio > 1
True
>>> S.GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> S.GoldenRatio.is_irrational
True
```

5.1.13 power

Pow

class sympy.core.power.Pow

Defines the expression x^y as “x raised to a power y”

Singleton definitions involving (0, 1, -1, oo, -oo, I, -I):

expr	value	reason
z^{**0}	1	Although arguments over 0^{**0} exist, see [2].
z^{**1}	z	
$(-\infty)^{**(-1)}$	0	
$(-1)^{**-1}$	-1	
$S.Zero^{**-1}$	zoo	This is not strictly true, as 0^{**-1} may be undefined, but is convenient in some contexts where the base is assumed to be positive.
1^{**-1}	1	
∞^{**-1}	0	
$0^{**\infty}$	0	Because for all complex numbers z near 0, $z^{**\infty} \rightarrow 0$.
$0^{**-\infty}$	zoo	This is not strictly true, as $0^{**\infty}$ may be oscillating between positive and negative values or rotating in the complex plane. It is convenient, however, when the base is positive.
$1^{**\infty}$ $1^{**-\infty}$	nan	Because there are various cases where $\lim(x(t),t)=1$, $\lim(y(t),t)=\infty$ (or $-\infty$), but $\lim(x(t)**y(t), t) \neq 1$. See [3].
$b^{**\infty}$	nan	Because b^{**z} has no limit as $z \rightarrow \infty$
$(-1)^{**\infty}$ $(-1)^{**(-\infty)}$	nan	Because of oscillations in the limit.
$\infty^{**\infty}$	∞	
$\infty^{**-\infty}$	0	
$(-\infty)^{**\infty}$ $(-\infty)^{**-\infty}$	nan	
∞^{**I} $(-\infty)^{**I}$	nan	∞^{**e} could probably be best thought of as the limit of x^{**e} for real x as x tends to ∞ . If e is I , then the limit does not exist and nan is used to indicate that.
$\infty^{**(1+I)}$ $(-\infty)^{**(1+I)}$	zoo	If the real part of e is positive, then the limit of $\text{abs}(x^{**e})$ is ∞ . So the limit value is zoo.
$\infty^{*(-1+I)}$ $-\infty^{*(-1+I)}$	0	If the real part of e is negative, then the limit is 0.

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754 conventions. This helps us avoid extra test-case code in the calculation of limits.

See also:

[sympy.core.numbers.Infinity](#) (page 147), [sympy.core.numbers.NegativeInfinity](#) (page 148), [sympy.core.numbers.NaN](#) (page 146)

References

[R70] (page 1853), [R71] (page 1853), [R72] (page 1853)

as_base_exp()

Return base and exp of self.

If base is 1/Integer, then return Integer, -exp. If this extra processing is not needed, the base and exp properties will give the raw arguments

Examples

```
>>> from sympy import Pow, S
>>> p = Pow(S.Half, 2, evaluate=False)
>>> p.as_base_exp()
(2, -2)
>>> p.args
(1/2, 2)
```

as_content_primitive(radical=False, clear=True)
Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import sqrt
>>> sqrt(4 + 4*sqrt(2)).as_content_primitive()
(2, sqrt(1 + sqrt(2)))
>>> sqrt(3 + 3*sqrt(2)).as_content_primitive()
(1, sqrt(3)*sqrt(1 + sqrt(2)))
```

```
>>> from sympy import expand_power_base, powsimp, Mul
>>> from sympy.abc import x, y
```

```
>>> ((2*x + 2)**2).as_content_primitive()
(4, (x + 1)**2)
>>> (4**((1 + y)/2)).as_content_primitive()
(2, 4**(y/2))
>>> (3**((1 + y)/2)).as_content_primitive()
(1, 3**((y + 1)/2))
>>> (3**((5 + y)/2)).as_content_primitive()
(9, 3**((y + 1)/2))
>>> eq = 3**(2 + 2*x)
>>> powsimp(eq) == eq
True
>>> eq.as_content_primitive()
(9, 3**(2*x))
>>> powsimp(Mul(*_))
3**(2*x + 2)
```

```
>>> eq = (2 + 2*x)**y
>>> s = expand_power_base(eq); s.is_Mul, s
(False, (2*x + 2)**y)
>>> eq.as_content_primitive()
(1, (2*(x + 1))**y)
>>> s = expand_power_base(_[1]); s.is_Mul, s
(True, 2**y*(x + 1)**y)
```

See docstring of Expr.as_content_primitive for more examples.

integer_nthroot

`sympy.core.power.integer_nthroot(y, n)`

Return a tuple containing $x = \text{floor}(y^{1/n})$ and a boolean indicating whether the result is exact (that is, whether $x^n == y$).

See also:

`sympy.ntheory.primetest.is_square`, `integer_log`

Examples

```
>>> from sympy import integer_nthroot
>>> integer_nthroot(16, 2)
(4, True)
>>> integer_nthroot(26, 2)
(5, False)
```

To simply determine if a number is a perfect square, the `is_square` function should be used:

```
>>> from sympy.ntheory.primetest import is_square
>>> is_square(26)
False
```

5.1.14 mul

Mul

`class sympy.core.mul.Mul`

`as_coeff_Mul(rational=False)`

Efficiently extract the coefficient of a product.

`as_coefficients_dict()`

Return a dictionary mapping terms to their coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. The dictionary is considered to have a single term.

Examples

```
>>> from sympy.abc import a, x
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> _[a]
0
```

`as_content_primitive(radical=False, clear=True)`

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import sqrt
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(-sqrt(2) + 1))
```

See docstring of `Expr.as_content_primitive` for more examples.

as_ordered_factors(order=None)

Transform an expression into an ordered list of factors.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x, y
```

```
>>> (2*x*y*sin(x)*cos(x)).as_ordered_factors()
[2, x, y, sin(x), cos(x)]
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use `self.args[0]`;
- if you want to process the arguments of the tail then use `self.as_coef_mul()` which gives the head and a tuple containing the arguments of the tail when treated as a Mul.
- if you want the coefficient when self is treated as an Add then use `self.as_coeff_add()[0]`

```
>>> from sympy.abc import x, y
>>> (3*x*y).as_two_terms()
(3, x*y)
```

classmethod flatten(seq)

Return commutative, noncommutative and order arguments by combining related terms.

Notes

- In an expression like `a*b*c`, python process this through sympy as `Mul(Mul(a, b), c)`. This can have undesirable consequences.
 - Sometimes terms are not combined as one would like: {c.f. <https://github.com/sympy/sympy/issues/4596>}

```
>>> from sympy import Mul, sqrt
>>> from sympy.abc import x, y, z
>>> 2*(x + 1) # this is the 2-arg Mul behavior
2*x + 2
>>> y*(x + 1)*2
2*y*(x + 1)
```

(continues on next page)

(continued from previous page)

```
>>> 2*(x + 1)*y # 2-arg result will be obtained first
y*(2*x + 2)
>>> Mul(2, x + 1, y) # all 3 args simultaneously processed
2*y*(x + 1)
>>> 2*((x + 1)*y) # parentheses can control this behavior
2*y*(x + 1)
```

Powers with compound bases may not find a single base to combine with unless all arguments are processed at once. Post-processing may be necessary in such cases. {c.f. <https://github.com/sympy/sympy/issues/5728>}

```
>>> a = sqrt(x*sqrt(y))
>>> a**3
(x*sqrt(y))**(3/2)
>>> Mul(a,a,a)
(x*sqrt(y))**(3/2)
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> _.subs(a.base, z).subs(z, a.base)
(x*sqrt(y))**(3/2)
```

- If more than two terms are being multiplied then all the previous terms will be re-processed for each new argument. So if each of `a`, `b` and `c` were `Mul` (page 155) expression, then `a*b*c` (or building up the product with `*=`) will process all the arguments of `a` and `b` twice: once when `a*b` is computed and again when `c` is multiplied.

Using `Mul(a, b, c)` will process all arguments once.

- The results of `Mul` are cached according to arguments, so `flatten` will only be called once for `Mul(a, b, c)`. If you can structure a calculation so the arguments are most likely to be repeats then this can save time in computing the answer. For example, say you had a `Mul`, `M`, that you wished to divide by `d[i]` and multiply by `n[i]` and you suspect there are many repeats in `n`. It would be better to compute `M*n[i]/d[i]` rather than `M/d[i]*n[i]` since every time `n[i]` is a repeat, the product, `M*n[i]` will be returned without flattening – the cached value will be returned. If you divide by the `d[i]` first (and those are more unique than the `n[i]`) then that will create a new `Mul`, `M/d[i]` the args of which will be traversed again when it is multiplied by `n[i]`.

{c.f. <https://github.com/sympy/sympy/issues/5706>}

This consideration is moot if the cache is turned off.

Nb

The validity of the above notes depends on the implementation details of `Mul` and `flatten` which may change at any time. Therefore, you should only consider them when your code is highly performance sensitive.

Removal of 1 from the sequence is already handled by `AssocOp.__new__`.

prod

```
sympy.core.mul.prod(a, start=1)
```

Return product of elements of a. Start with int 1 so if only ints are included then an int result is returned.

Examples

```
>>> from sympy import prod, S
>>> prod(range(3))
0
>>> type(_) is int
True
>>> prod([S(2), 3])
6
>>> _.is_Integer
True
```

You can start the product at something other than 1:

```
>>> prod([1, 2], 3)
6
```

5.1.15 add

Add

class sympy.core.add.Add

as_coeff_Add(rational=False)
Efficiently extract the coefficient of a summation.

as_coeff_add(*deps)
Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.

Examples

```
>>> from sympy.abc import x
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

as_coefficients_dict()
Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```
>>> from sympy.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

as_content_primitive(radical=False, clear=True)

Return the tuple (R, self/R) where R is the positive Rational extracted from self. If radical is True (default is False) then common radicals will be removed and included as a factor of the primitive expression.

Examples

```
>>> from sympy import sqrt
>>> (3 + 3*sqrt(2)).as_content_primitive()
(3, 1 + sqrt(2))
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

See docstring of Expr.as_content_primitive for more examples.

as_real_imag(deep=True, **hints)

returns a tuple representing a complex number

Examples

```
>>> from sympy import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as_coef_add() which gives the head and a tuple containing the arguments of the tail when treated as an Add.
- if you want the coefficient when self is treated as a Mul then use self.as_coeff_mul()[0]

```
>>> from sympy.abc import x, y
>>> (3*x - 2*y + 5).as_two_terms()
(5, 3*x - 2*y)
```

classmethod class_key()

Nice order of classes

extract_leading_order(symbols, point=None)

Returns the leading term and its order.

Examples

```
>>> from sympy.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), 0(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, 0(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, 0(x)),)
```

classmethod flatten(seq)

Takes the sequence “seq” of nested Adds and returns a flatten list.

Returns: (commutative_part, noncommutative_part, order_symbols)

Applies associativity, all terms are commutable with respect to addition.

NB: the removal of 0 is already handled by AssocOp.__new__

See also:

[sympy.core.mul.Mul.flatten](#) (page 156)

primitive()

Return $(R, \text{self}/R)$ where R' is the Rational GCD of self' .

R is collected only from the leading coefficient of each term.

Examples

```
>>> from sympy.abc import x, y
```

```
>>> (2*x + 4*y).primitive()
(2, x + 2*y)
```

```
>>> (2*x/3 + 4*y/9).primitive()
(2/9, 3*x + 2*y)
```

```
>>> (2*x/3 + 4.2*y).primitive()
(1/3, 2*x + 12.6*y)
```

No subprocessing of term factors is performed:

```
>>> ((2 + 2*x)*x + 2).primitive()
(1, x*(2*x + 2) + 2)
```

Recursive processing can be done with the `as_content_primitive()` method:

```
>>> ((2 + 2*x)*x + 2).as_content_primitive()
(2, x*(x + 1) + 1)
```

See also: `primitive()` function in `polytools.py`

5.1.16 mod

Mod

class `sympy.core.mod.Mod`

Represents a modulo operation on symbolic expressions.

Receives two arguments, dividend `p` and divisor `q`.

The convention used is the same as Python's: the remainder always has the same sign as the divisor.

Examples

```
>>> from sympy.abc import x, y
>>> x**2 % y
Mod(x**2, y)
>>> _.subs({x: 5, y: 6})
1
```

5.1.17 relational

Rel

`sympy.core.relational.Rel`

alias of `sympy.core.relational.Relational`

Eq

`sympy.core.relational.Eq`

alias of `sympy.core.relational.Equality` (page 162)

Ne

`sympy.core.relational.Ne`

alias of `sympy.core.relational.Unequality` (page 169)

Lt

`sympy.core.relational.Lt`

alias of `sympy.core.relational.StrictLessThan` (page 172)

Le

`sympy.core.relational.Le`

alias of `sympy.core.relational.LessThan` (page 166)

Gt

`sympy.core.relational.Gt`

alias of `sympy.core.relational.StrictGreaterThan` (page 169)

Ge

`sympy.core.relational.Ge`

alias of `sympy.core.relational.GreaterThan` (page 163)

Equality

class `sympy.core.relational.Equality`

An equal relation between two objects.

Represents that two objects are equal. If they can be easily shown to be definitively equal (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained as an unevaluated Equality object. Use the `simplify` function on this object for more nontrivial evaluation of the equality relation.

As usual, the keyword argument `evaluate=False` can be used to prevent any evaluation.

See also:

`sympy.logic.boolalg.Equivalent` (page 725) for representing equality between two boolean expressions

Notes

This class is not the same as the `==` operator. The `==` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an `evalEq` method, it can be used in place of the default algorithm. If `lhs.evalEq(rhs)` or `rhs.evalEq(lhs)` returns anything other than None, that return value will be substituted for the Equality. If None is returned by `evalEq`, an Equality object will be created as usual.

Examples

```
>>> from sympy import Eq, simplify, exp, cos
>>> from sympy.abc import x, y
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
False
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> _.doit()
False
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(exp(x), sinh(x) + cosh(x))
>>> simplify(_)
True
```


GreaterThan

`class sympy.core.relational.GreaterThan`

Class representations of inequalities.

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
<code>GreaterThan</code>	$(\geq)$
<code>LessThan</code>	$(<=)$
<code>StrictGreaterThan</code>	$(>)$
<code>StrictLessThan</code>	$(<)$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math equivalent
<code>GreaterThan(lhs, rhs)</code>	$\text{lhs} \geq \text{rhs}$
<code>LessThan(lhs, rhs)</code>	$\text{lhs} <= \text{rhs}$
<code>StrictGreaterThan(lhs, rhs)</code>	$\text{lhs} > \text{rhs}$
<code>StrictLessThan(lhs, rhs)</code>	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of `Relations`, `*Than` inequality objects also have the `.lhs` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lhs` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lhs, e.lhs, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “`1 < x`” is evaluated, Python recognizes that the number 1 is a native number and that `x` is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “`x > 1`” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

1. “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

- (2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving ‘==’ or ‘!=’. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R73] (page 1853), there is no way for SymPy to create a chained inequality with that syntax so one must use `And`:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

LessThan

`class sympy.core.relational.LessThan`

Class representations of inequalities.

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
<code>GreaterThan</code>	$(\geq)$
<code>LessThan</code>	$(<=)$
<code>StrictGreaterThan</code>	$(>)$
<code>StrictLessThan</code>	$(<)$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math equivalent
<code>GreaterThan(lhs, rhs)</code>	$\text{lhs} \geq \text{rhs}$
<code>LessThan(lhs, rhs)</code>	$\text{lhs} \leq \text{rhs}$
<code>StrictGreaterThan(lhs, rhs)</code>	$\text{lhs} > \text{rhs}$
<code>StrictLessThan(lhs, rhs)</code>	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of `Relations`, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “ $1 < x$ ” is evaluated, Python recognizes that the number 1 is a native number and that x is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “ $x > 1$ ” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

1. “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

- (2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving ‘==’ or ‘!=’. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R74] (page 1854), there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Unequality

class sympy.core.relational.Unequality

An unequal relation between two objects.

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to False; if definitively unequal, this will reduce to True. Otherwise, the relation is maintained as an Unequality object.

See also:

[Equality](#) (page 162)

Notes

This class is not the same as the `!=` operator. The `!=` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of Equality. As such, it uses the same algorithms, including any available `evalEq` methods.

Examples

```
>>> from sympy import Ne
>>> from sympy.abc import x, y
>>> Ne(y, x+x**2)
Ne(y, x**2 + x)
```

StrictGreaterThan

class sympy.core.relational.StrictGreaterThan

Class representations of inequalities.

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	$(\geq)$
LessThan	$(\leq)$
StrictGreaterThan	$(>)$
StrictLessThan	$(<)$

All classes take two arguments, lhs and rhs.

Signature	Example	Math equivalent
GreaterThan	(lhs, rhs)	$\text{lhs} \geq \text{rhs}$
LessThan	(lhs, rhs)	$\text{lhs} \leq \text{rhs}$
StrictGreaterThan	(lhs, rhs)	$\text{lhs} > \text{rhs}$
StrictLessThan	(lhs, rhs)	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lhs` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lhs` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lhs, e.lhs, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “`1 < x`” is evaluated, Python recognizes that the number 1 is a native number and that `x` is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “`x > 1`” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

1. “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

- (2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```


The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving '==' or '!='. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [\[R75\]](#) (page 1854), there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the '&' operator, it cannot be done with the 'and' operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

StrictLessThan

class `sympy.core.relational.StrictLessThan`

Class representations of inequalities.

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
<code>GreaterThan</code>	$(\geq)$
<code>LessThan</code>	$(\leq)$
<code>StrictGreaterThan</code>	$(>)$
<code>StrictLessThan</code>	$(<)$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	$\text{lhs} \geq \text{rhs}$
LessThan(lhs, rhs)	$\text{lhs} \leq \text{rhs}$
StrictGreaterThan(lhs, rhs)	$\text{lhs} > \text{rhs}$
StrictLessThan(lhs, rhs)	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what you write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “`1 < x`” is evaluated, Python recognizes that the number 1 is a native number and that `x` is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “`x > 1`” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

1. “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

- (2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e # neither side is a literal
True
>>> e == x < 1 # expecting True, too
False
>>> e != x < 1 # expecting False
x < 1
>>> x < 1 != x < 1 # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving ‘==’ or ‘!=’. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [\[R76\]](#) (page 1855), there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]: # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ($\geq$, $>$, $\leq$, $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

5.1.18 multidimensional

vectorize

class `sympy.core.multidimensional.vectorize(*mdargs)`

Generalizes a function taking scalars to accept multidimensional arguments.

For example

```
>>> from sympy import diff, sin, symbols, Function
>>> from sympy.core.multidimensional import vectorize
>>> x, y, z = symbols('x y z')
>>> f, g, h = list(map(Function, 'fgh'))
```

```
>>> @vectorize(0)
... def vsin(x):
...     return sin(x)
```

```
>>> vsin([1, x, y])
[sin(1), sin(x), sin(y)]
```

```
>>> @vectorize(0, 1)
... def vdiff(f, y):
...     return diff(f, y)
```

```
>>> vdiff([f(x, y, z), g(x, y, z), h(x, y, z)], [x, y, z])
[[Derivative(f(x, y, z), x), Derivative(f(x, y, z), y), Derivative(f(x, y, z),
↪z)], [Derivative(g(x, y, z), x), Derivative(g(x, y, z), y), Derivative(g(x, y,
↪z), z)], [Derivative(h(x, y, z), x), Derivative(h(x, y, z), y), Derivative(h(x,
↪y, z), z)]]
```

5.1.19 function

Lambda

class sympy.core.function.Lambda

Lambda(x, expr) represents a lambda function similar to Python's 'lambda x: expr'. A function of several variables is written as Lambda((x, y, ...), expr).

A simple example:

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> f = Lambda(x, x**2)
>>> f(4)
16
```

For multivariate functions, use:

```
>>> from sympy.abc import y, z, t
>>> f2 = Lambda((x, y, z, t), x + y**z + t**z)
>>> f2(1, 2, 3, 4)
73
```

A handy shortcut for lots of arguments:

```
>>> p = x, y, z
>>> f = Lambda(p, x + y*z)
>>> f(*p)
x + y*z
```

expr

The return value of the function

is_identity

Return True if this Lambda is an identity function.

variables

The variables used in the internal representation of the function

WildFunction

class sympy.core.function.WildFunction(name, **assumptions)

A WildFunction function matches any function (with its arguments).

Examples

```
>>> from sympy import WildFunction, Function, cos
>>> from sympy.abc import x, y
>>> F = WildFunction('F')
>>> f = Function('f')
>>> F.nargs
Naturals0
>>> x.match(F)
>>> F.match(F)
{F_: F_}
>>> f(x).match(F)
{F_: f(x)}
>>> cos(x).match(F)
{F_: cos(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a given number of arguments, set `nargs` to the desired value at instantiation:

```
>>> F = WildFunction('F', nargs=2)
>>> F.nargs
{2}
>>> f(x).match(F)
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a range of arguments, set `nargs` to a tuple containing the desired number of arguments, e.g. if `nargs = (1, 2)` then functions with 1 or 2 arguments will be matched.

```
>>> F = WildFunction('F', nargs=(1, 2))
>>> F.nargs
{1, 2}
>>> f(x).match(F)
{F_: f(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
>>> f(x, y, 1).match(F)
```

Derivative

class sympy.core.function.Derivative

Carries out differentiation of the given expression with respect to symbols.

`expr` must define `._eval_derivative(symbol)` method that returns the differentiation result. This function only needs to consider the non-trivial case where `expr` contains `symbol` and it should call the `diff()` method internally (not `._eval_derivative`); `Derivative` should be the only one to call `._eval_derivative`.

Simplification of high-order derivatives:

Because there can be a significant amount of simplification that can be done when multiple differentiations are performed, results will be automatically simplified in a fairly conservative fashion unless the keyword `simplify` is set to `False`.

```
>>> from sympy import sqrt, diff
>>> from sympy.abc import x
>>> e = sqrt((x + 1)**2 + x)
>>> diff(e, (x, 5), simplify=False).count_ops()
136
>>> diff(e, (x, 5)).count_ops()
30
```

Ordering of variables:

If `evaluate` is set to `True` and the expression can not be evaluated, the list of differentiation symbols will be sorted, that is, the expression is assumed to have continuous derivatives up to the order asked. This sorting assumes that derivatives wrt Symbols commute, derivatives wrt non-Symbols commute, but Symbol and non-Symbol derivatives don't commute with each other.

Derivative wrt non-Symbols:

This class also allows derivatives wrt non-Symbols that have `_diff_wrt` set to `True`, such as `Function` and `Derivative`. When a derivative wrt a non-Symbol is attempted, the non-Symbol is temporarily converted to a Symbol while the differentiation is performed.

Note that this may seem strange, that `Derivative` allows things like `f(g(x)).diff(g(x))`, or even `f(cos(x)).diff(cos(x))`. The motivation for allowing this syntax is to make it easier to work with variational calculus (i.e., the Euler-Lagrange method). The best way to understand this is that the action of derivative with respect to a non-Symbol is defined by the above description: the object is substituted for a Symbol and the derivative is taken with respect to that. This action is only allowed for objects for which this can be done unambiguously, for example `Function` and `Derivative` objects. Note that this leads to what may appear to be mathematically inconsistent results. For example:

```
>>> from sympy import cos, sin, sqrt
>>> from sympy.abc import x
>>> (2*cos(x)).diff(cos(x))
2
>>> (2*sqrt(1 - sin(x)**2)).diff(cos(x))
0
```

This appears wrong because in fact `2*cos(x)` and `2*sqrt(1 - sin(x)**2)` are identically equal. However this is the wrong way to think of this. Think of it instead as if we have something like this:

```
>>> from sympy.abc import c, s, u, x
>>> def F(u):
...     return 2*u
...
>>> def G(u):
...     return 2*sqrt(1 - u**2)
...
>>> F(cos(x))
2*cos(x)
>>> G(sin(x))
2*sqrt(-sin(x)**2 + 1)
```

(continues on next page)

(continued from previous page)

```
>>> F(c).diff(c)
2
>>> F(cos(x)).diff(cos(x))
2
>>> G(s).diff(c)
0
>>> G(sin(x)).diff(cos(x))
0
```

Here, the Symbols c and s act just like the functions $\cos(x)$ and $\sin(x)$, respectively. Think of $2*\cos(x)$ as $f(c).subs(c, \cos(x))$ (or $f(c)$ at $c = \cos(x)$) and $2*\sqrt{1 - \sin(x)**2}$ as $g(s).subs(s, \sin(x))$ (or $g(s)$ at $s = \sin(x)$), where $f(u) == 2*u$ and $g(u) == 2*\sqrt{1 - u**2}$. Here, we define the function first and evaluate it at the function, but we can actually unambiguously do this in reverse in SymPy, because `expr.subs(Function, Symbol)` is well-defined: just structurally replace the function everywhere it appears in the expression.

This is the same notational convenience used in the Euler-Lagrange method when one says $F(t, f(t), f'(t)).diff(f(t))$. What is actually meant is that the expression in question is represented by some $F(t, u, v)$ at $u = f(t)$ and $v = f'(t)$, and $F(t, f(t), f'(t)).diff(f(t))$ simply means $F(t, u, v).diff(u)$ at $u = f(t)$.

We do not allow derivatives to be taken with respect to expressions where this is not so well defined. For example, we do not allow `expr.diff(x*y)` because there are multiple ways of structurally defining where $x*y$ appears in an expression, some of which may surprise the reader (for example, a very strict definition would have that $(x*y*z).diff(x*y) == 0$).

```
>>> from sympy.abc import x, y, z
>>> (x*y*z).diff(x*y)
Traceback (most recent call last):
...
ValueError: Can't differentiate wrt the variable: x*y, 1
```

Note that this definition also fits in nicely with the definition of the chain rule. Note how the chain rule in SymPy is defined using unevaluated Subs objects:

```
>>> from sympy import symbols, Function
>>> f, g = symbols('f g', cls=Function)
>>> f(2*g(x)).diff(x)
2*Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                           (_xi_1,), (2*g(x),))

>>> f(g(x)).diff(x)
Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                           (_xi_1,), (g(x),))
```

Finally, note that, to be consistent with variational calculus, and to ensure that the definition of substituting a Function for a Symbol in an expression is well-defined, derivatives of functions are assumed to not be related to the function. In other words, we have:

```
>>> from sympy import diff
>>> diff(f(x), x).diff(f(x))
0
```

The same is true for derivatives of different orders:

```
>>> diff(f(x), x, 2).diff(diff(f(x), x, 1))
0
>>> diff(f(x), x, 1).diff(diff(f(x), x, 2))
0
```

Note, any class can allow derivatives to be taken with respect to itself. See the docstring of `Expr.diff_wrt`.

Examples

Some basic examples:

```
>>> from sympy import Derivative, Symbol, Function
>>> f = Function('f')
>>> g = Function('g')
>>> x = Symbol('x')
>>> y = Symbol('y')
```

```
>>> Derivative(x**2, x, evaluate=True)
2*x
>>> Derivative(Derivative(f(x,y), x), y)
Derivative(f(x, y), x, y)
>>> Derivative(f(x), x, 3)
Derivative(f(x), (x, 3))
>>> Derivative(f(x, y), y, x, evaluate=True)
Derivative(f(x, y), x, y)
```

Now some derivatives wrt functions:

```
>>> Derivative(f(x)**2, f(x), evaluate=True)
2*f(x)
>>> Derivative(f(g(x)), x, evaluate=True)
Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                        (_xi_1,), (g(x),))
```

as_finite_difference(points=1, x0=None, wrt=None)
Expresses a `Derivative` instance as a finite difference.

Parameters **points** : sequence or coefficient, optional

If sequence: discrete values (length $\geq$ order+1) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length order+1 centered around x0. Default: 1 (step-size 1)

x0 : number or Symbol, optional

the value of the independent variable (wrt) at which the derivative is to be approximated. Default: same as wrt.

wrt : Symbol, optional

“with respect to” the variable for which the (partial) derivative is to be approximated for. If not provided it is required that the derivative is ordinary. Default: None.

See also:

`sympy.calculus.finite_diff.apply_finite_diff` (page 1464), `sympy.calculus.finite_diff.differentiate_finite` (page 1466), `sympy.calculus.finite_diff.finite_diff_weights` (page 1467)

Examples

```
>>> from sympy import symbols, Function, exp, sqrt, Symbol
>>> x, h = symbols('x h')
>>> f = Function('f')
>>> f(x).diff(x).as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

The default step size and number of points are 1 and order + 1 respectively. We can change the step size by passing a symbol as a parameter:

```
>>> f(x).diff(x).as_finite_difference(h)
-f(-h/2 + x)/h + f(h/2 + x)/h
```

We can also specify the discretized values to be used in a sequence:

```
>>> f(x).diff(x).as_finite_difference([x, x+h, x+2*h])
-3*f(x)/(2*h) + 2*f(h + x)/h - f(2*h + x)/(2*h)
```

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around x_0 , but we can get an expression estimating the derivative at an offset:

```
>>> e, sq2 = exp(1), sqrt(2)
>>> xl = [x-h, x+h, x+e*h]
>>> f(x).diff(x, 1).as_finite_difference(xl, x+h*sq2)
2*h*((h + sqrt(2)*h)/(2*h) - (-sqrt(2)*h + h)/(2*h))*f(E*h + x)/...
```

Partial derivatives are also supported:

```
>>> y = Symbol('y')
>>> d2fdxdy=f(x,y).diff(x,y)
>>> d2fdxdy.as_finite_difference(wrt=x)
-Derivative(f(x - 1/2, y), y) + Derivative(f(x + 1/2, y), y)
```

We can apply `as_finite_difference` to `Derivative` instances in compound expressions using `replace`:

```
>>> (1 + 42*f(x).diff(x)).replace(lambda arg: arg.is_Derivative,
... lambda arg: arg.as_finite_difference())
42*(-f(x - 1/2) + f(x + 1/2)) + 1
```

`doit_numerically(z0)`

Evaluate the derivative at z numerically.

When we can represent derivatives at a point, this should be folded into the normal `evalf`. For now, we need a special method.

diff

`sympy.core.function.diff(f, *symbols, **kwargs)`
Differentiate f with respect to $symbols$.

This is just a wrapper to unify `.diff()` and the `Derivative` class; its interface is similar to that of `integrate()`. You can use the same shortcuts for multiple variables as with `Derivative`. For example, `diff(f(x), x, x, x)` and `diff(f(x), x, 3)` both return the third derivative of $f(x)$.

You can pass `evaluate=False` to get an unevaluated `Derivative` class. Note that if there are 0 symbols (such as `diff(f(x), x, 0)`), then the result will be the function (the zeroth derivative), even if `evaluate=False`.

See also:

[Derivative](#) (page 177)

`sympy.geometry.util.idiff` computes the derivative implicitly

References

http://reference.wolfram.com/legacy/v5_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html

Examples

```
>>> from sympy import sin, cos, Function, diff
>>> from sympy.abc import x, y
>>> f = Function('f')
```

```
>>> diff(sin(x), x)
cos(x)
>>> diff(f(x), x, x, x)
Derivative(f(x), (x, 3))
>>> diff(f(x), x, 3)
Derivative(f(x), (x, 3))
>>> diff(sin(x)*cos(y), x, 2, y, 2)
sin(x)*cos(y)
```

```
>>> type(diff(sin(x), x))
cos
>>> type(diff(sin(x), x, evaluate=False))
<class 'sympy.core.function.Derivative'>
>>> type(diff(sin(x), x, 0))
sin
>>> type(diff(sin(x), x, 0, evaluate=False))
sin
```

```
>>> diff(sin(x))
cos(x)
>>> diff(sin(x*y))
Traceback (most recent call last):
...
ValueError: specify differentiation variables to differentiate sin(x*y)
```

Note that `diff(sin(x))` syntax is meant only for convenience in interactive sessions and should be avoided in library code.

FunctionClass

class sympy.core.function.**FunctionClass**(*args, **kwargs)

Base class for function classes. FunctionClass is a subclass of type.

Use Function('<function name>' [, signature]) to create undefined function classes.

nargs

Return a set of the allowed number of arguments for the function.

Examples

```
>>> from sympy.core.function import Function
>>> from sympy.abc import x, y
>>> f = Function('f')
```

If the function can take any number of arguments, the set of whole numbers is returned:

```
>>> Function('f').nargs
Naturals0
```

If the function was initialized to accept one or more arguments, a corresponding set will be returned:

```
>>> Function('f', nargs=1).nargs
{1}
>>> Function('f', nargs=(2, 1)).nargs
{1, 2}
```

The undefined function, after application, also has the nargs attribute; the actual number of arguments is always available by checking the args attribute:

```
>>> f = Function('f')
>>> f(1).nargs
Naturals0
>>> len(f(1).args)
1
```

Function

class sympy.core.function.**Function**

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

Examples

First example shows how to use Function as a constructor for undefined function classes:

```
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
```

(continues on next page)

(continued from previous page)

```

>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)

```

Assumptions can be passed to Function.

```

>>> f_real = Function('f', real=True)
>>> f_real(x).is_real
True

```

Note that assumptions on a function are unrelated to the assumptions on the variable it is called on. If you want to add a relationship, subclass Function and define the appropriate `_eval_is_assumption` methods.

In the following example Function is used as a base class for `my_func` that represents a mathematical function `my_func`. Suppose that it is well known, that `my_func(0)` is 1 and `my_func` at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that `my_func(x)` is real exactly when `x` is real. Here is an implementation that honours those requirements:

```

>>> from sympy import Function, S, oo, I, sin
>>> class my_func(Function):
...
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x is S.Zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...
...     def _eval_is_real(self):
...         return self.args[0].is_real
...
>>> x = S('x')
>>> my_func(0) + sin(0)
1
>>> my_func(oo)
0
>>> my_func(3.54).n() # Not yet implemented for my_func.
my_func(3.54)
>>> my_func(I).is_real
False

```

In order for `my_func` to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then `nargs` must be defined, e.g. if `my_func` can take one or two arguments then,

```
>>> class my_func(Function):
...     nargs = (1, 2)
...
>>>
```

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

fdiff(argindex=1)

Returns the first derivative of the function.

is_commutative

Returns whether the function is commutative.

Note: Not all functions are the same

SymPy defines many functions (like `cos` and `factorial`). It also allows the user to create generic functions which act as argument holders. Such functions are created just like symbols:

```
>>> from sympy import Function, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> f(2) + f(x)
f(2) + f(x)
```

If you want to see which functions appear in an expression you can use the `atoms` method:

```
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function)
{f(x), cos(x)}
```

If you just want the function you defined, not SymPy functions, the thing to search for is `AppliedUndef`:

```
>>> from sympy.core.function import AppliedUndef
>>> e.atoms(AppliedUndef)
{f(x)}
```

Subs

class `sympy.core.function.Subs`

Represents unevaluated substitutions of an expression.

`Subs(expr, x, x0)` receives 3 arguments: an expression, a variable or list of distinct variables and a point or list of evaluation points corresponding to those variables.

`Subs` objects are generally useful to represent unevaluated derivatives calculated at a point.

The variables may be expressions, but they are subjected to the limitations of `subs()`, so it is usually a good practice to use only symbols for variables, since in that case there can be no ambiguity.

There's no automatic expansion - use the method `.doit()` to effect all possible substitutions of the object and also of objects inside the expression.

When evaluating derivatives at a point that is not a symbol, a Subs object is returned. One is also able to calculate derivatives of Subs objects - in this case the expression is always expanded (for the unevaluated form, use Derivative()).

A simple example:

```
>>> from sympy import Subs, Function, sin
>>> from sympy.abc import x, y, z
>>> f = Function('f')
>>> e = Subs(f(x).diff(x), x, y)
>>> e.subs(y, 0)
Subs(Derivative(f(x), x), (x,), (0,))
>>> e.subs(f, sin).doit()
cos(y)
```

An example with several variables:

```
>>> Subs(f(x)*sin(y) + z, (x, y), (0, 1))
Subs(z + f(x)*sin(y), (x, y), (0, 1))
>>> .doit()
z + f(0)*sin(1)
```

expr

The expression on which the substitution operates

point

The values for which the variables are to be substituted

variables

The variables to be evaluated

expand

`sympy.core.function.expand(e, deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints)`

Expand an expression using methods given as hints.

Hints evaluated unless explicitly set to False are: `basic`, `log`, `multinomial`, `mul`, `power_base`, and `power_exp`. The following hints are supported but not applied unless set to True: `complex`, `func`, and `trig`. In addition, the following meta-hints are supported by some or all of the other hints: `frac`, `numer`, `denom`, `modulus`, and `force`. `deep` is supported by all hints. Additionally, subclasses of `Expr` may define their own hints or meta-hints.

The `basic` hint is used for any special rewriting of an object that should be done automatically (along with the other hints like `mul`) when `expand` is called. This is a catch-all hint to handle any sort of expansion that may not be described by the existing hint names. To use this hint an object should override the `_eval_expand_basic` method. Objects may also define their own `expand` methods, which are not run by default. See the API section below.

If `deep` is set to True (the default), things like arguments of functions are recursively expanded. Use `deep=False` to only expand on the top level.

If the `force` hint is used, assumptions about variables will be ignored in making the expansion.

See also:

[expand_log](#) (page 193), [expand_mul](#) (page 193), [expand_multinomial](#) (page 194), [expand_complex](#) (page 194), [expand_trig](#) (page 194), [expand_power_base](#) (page 195), [expand_power_exp](#) (page 195), [expand_func](#) (page 193), [hyperexpand](#)

Notes

- You can shut off unwanted methods:

```
>>> (exp(x + y)*(x + y)).expand()
x*exp(x)*exp(y) + y*exp(x)*exp(y)
>>> (exp(x + y)*(x + y)).expand(power_exp=False)
x*exp(x + y) + y*exp(x + y)
>>> (exp(x + y)*(x + y)).expand(mul=False)
(x + y)*exp(x)*exp(y)
```

- Use `deep=False` to only expand on the top level:

```
>>> exp(x + exp(x + y)).expand()
exp(x)*exp(exp(x)*exp(y))
>>> exp(x + exp(x + y)).expand(deep=False)
exp(x)*exp(exp(x + y))
```

- Hints are applied in an arbitrary, but consistent order (in the current implementation, they are applied in alphabetical order, except `multinomial` comes before `mul`, but this may change). Because of this, some hints may prevent expansion by other hints if they are applied first. For example, `mul` may distribute multiplications and prevent `log` and `power_base` from expanding them. Also, if `mul` is applied before `multinomial`, the expression might not be fully distributed. The solution is to use the various ‘`expand_hint`’ helper functions or to use `hint=False` to this function to finely control which hints are applied. Here are some examples:

```
>>> from sympy import expand, expand_mul, expand_power_base
>>> x, y, z = symbols('x,y,z', positive=True)

>>> expand(log(x*(y + z)))
log(x) + log(y + z)
```

Here, we see that `log` was applied before `mul`. To get the `mul` expanded form, either of the following will work:

```
>>> expand_mul(log(x*(y + z)))
log(x*y + x*z)
>>> expand(log(x*(y + z)), log=False)
log(x*y + x*z)
```

A similar thing can happen with the `power_base` hint:

```
>>> expand((x*(y + z))**x)
(x*y + x*z)**x
```

To get the `power_base` expanded form, either of the following will work:

```
>>> expand((x*(y + z))**x, mul=False)
x**x*(y + z)**x
>>> expand_power_base((x*(y + z))**x)
```

(continues on next page)

(continued from previous page)

```
x**x*(y + z)**x

>>> expand((x + y)*y/x)
y + y**2/x
```

The parts of a rational expression can be targeted:

```
>>> expand((x + y)*y/x/(x + 1), frac=True)
(x*y + y**2)/(x**2 + x)
>>> expand((x + y)*y/x/(x + 1), numer=True)
(x*y + y**2)/(x*(x + 1))
>>> expand((x + y)*y/x/(x + 1), denom=True)
y*(x + y)/(x**2 + x)
```

- The modulus meta-hint can be used to reduce the coefficients of an expression post-expansion:

```
>>> expand((3*x + 1)**2)
9*x**2 + 6*x + 1
>>> expand((3*x + 1)**2, modulus=5)
4*x**2 + x + 1
```

- Either `expand()` the function or `.expand()` the method can be used. Both are equivalent:

```
>>> expand((x + 1)**2)
x**2 + 2*x + 1
>>> ((x + 1)**2).expand()
x**2 + 2*x + 1
```

Examples

```
>>> from sympy import Expr, sympify
>>> class MyClass(Expr):
...     def __new__(cls, *args):
...         args = sympify(args)
...         return Expr.__new__(cls, *args)
...
...     def _eval_expand_double(self, **hints):
...         """
...         Doubles the args of MyClass.
...
...         If there more than four args, doubling is not performed,
...         unless force=True is also used (False by default).
...         """
...         force = hints.pop('force', False)
...         if not force and len(self.args) > 4:
...             return self
...         return self.func(*(self.args + self.args))
...
>>> a = MyClass(1, 2, MyClass(3, 4))
>>> a
MyClass(1, 2, MyClass(3, 4))
>>> a.expand(double=True)
```

(continues on next page)

(continued from previous page)

```
MyClass(1, 2, MyClass(3, 4, 3, 4), 1, 2, MyClass(3, 4, 3, 4))
>>> a.expand(double=True, deep=False)
MyClass(1, 2, MyClass(3, 4), 1, 2, MyClass(3, 4))
```

```
>>> b = MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True)
MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True, force=True)
MyClass(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

Hints

These hints are run by default

Mul

Distributes multiplication over addition:

```
>>> from sympy import cos, exp, sin
>>> from sympy.abc import x, y, z
>>> (y*(x + z)).expand(mul=True)
x*y + y*z
```

Multinomial

Expand $(x + y + \dots)^n$ where n is a positive integer.

```
>>> ((x + y + z)**2).expand(multinomial=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Power_exp

Expand addition in exponents into multiplied bases.

```
>>> exp(x + y).expand(power_exp=True)
exp(x)*exp(y)
>>> (2**(x + y)).expand(power_exp=True)
2**x*2**y
```

Power_base

Split powers of multiplied bases.

This only happens by default if assumptions allow, or if the `force` meta-hint is used:

```
>>> ((x*y)**z).expand(power_base=True)
(x*y)**z
>>> ((x*y)**z).expand(power_base=True, force=True)
x**z*y**z
>>> ((2*y)**z).expand(power_base=True)
2**z*y**z
```

Note that in some cases where this expansion always holds, SymPy performs it automatically:

```
>>> (x*y)**2
x**2*y**2
```

Log

Pull out power of an argument as a coefficient and split logs products into sums of logs.

Note that these only work if the arguments of the log function have the proper assumptions—the arguments must be positive and the exponents must be real—or else the force hint must be True:

```
>>> from sympy import log, symbols
>>> log(x**2*y).expand(log=True)
log(x**2*y)
>>> log(x**2*y).expand(log=True, force=True)
2*log(x) + log(y)
>>> x, y = symbols('x,y', positive=True)
>>> log(x**2*y).expand(log=True)
2*log(x) + log(y)
```

Basic

This hint is intended primarily as a way for custom subclasses to enable expansion by default.

These hints are not run by default:

Complex

Split an expression into real and imaginary parts.

```
>>> x, y = symbols('x,y')
>>> (x + y).expand(complex=True)
re(x) + re(y) + I*im(x) + I*im(y)
>>> cos(x).expand(complex=True)
-I*sin(re(x))*sinh(im(x)) + cos(re(x))*cosh(im(x))
```

Note that this is just a wrapper around `as_real_imag()`. Most objects that wish to redefine `_eval_expand_complex()` should consider redefining `as_real_imag()` instead.

Func

Expand other functions.

```
>>> from sympy import gamma
>>> gamma(x + 1).expand(func=True)
x*gamma(x)
```

Trig

Do trigonometric expansions.

```
>>> cos(x + y).expand(trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)
>>> sin(2*x).expand(trig=True)
2*sin(x)*cos(x)
```

Note that the forms of $\sin(n*x)$ and $\cos(n*x)$ in terms of $\sin(x)$ and $\cos(x)$ are not unique, due to the identity $\sin^2(x) + \cos^2(x) = 1$. The current implementation uses the form obtained from Chebyshev polynomials, but this may change. See [this MathWorld article](#) for more information.

Api

Objects can define their own expand hints by defining `_eval_expand_hint()`. The function should take the form:

```
def _eval_expand_hint(self, **hints):
    # Only apply the method to the top-level expression
    ...
```

See also the example below. Objects should define `_eval_expand_hint()` methods only if hint applies to that specific object. The generic `_eval_expand_hint()` method defined in `Expr` will handle the no-op case.

Each hint should be responsible for expanding that hint only. Furthermore, the expansion should be applied to the top-level expression only. `expand()` takes care of the recursion that happens when `deep=True`.

You should only call `_eval_expand_hint()` methods directly if you are 100% sure that the object has the method, as otherwise you are liable to get unexpected `AttributeError`'s. Note, again, that you do not need to recursively apply the hint to args of your object: this is handled automatically by `expand()`. `_eval_expand_hint()` should generally not be used at all outside of an `_eval_expand_hint()` method. If you want to apply a specific expansion from within another method, use the public `expand()` function, method, or `expand_hint()` functions.

In order for `expand` to work, objects must be rebuildable by their args, i.e., `obj.func(*obj.args) == obj` must hold.

Expand methods are passed `**hints` so that expand hints may use 'metahints'-hints that control how different expand methods are applied. For example, the `force=True` hint described above that causes `expand(log=True)` to ignore assumptions is such a metahint. The deep meta-hint is handled exclusively by `expand()` and is not passed to `_eval_expand_hint()` methods.

Note that expansion hints should generally be methods that perform some kind of ‘expansion’. For hints that simply rewrite an expression, use the `.rewrite()` API.

PoleError

`class sympy.core.function.PoleError`

count_ops

`sympy.core.function.count_ops(expr, visual=False)`

Return a representation (integer or expression) of the operations in `expr`.

If `visual` is `False` (default) then the sum of the coefficients of the visual expression will be returned.

If `visual` is `True` then the number of each type of operation is shown with the core class types (or their virtual equivalent) multiplied by the number of times they occur.

If `expr` is an iterable, the sum of the op counts of the items will be returned.

Examples

```
>>> from sympy.abc import a, b, x, y
>>> from sympy import sin, count_ops
```

Although there isn’t a SUB object, minus signs are interpreted as either negations or subtractions:

```
>>> (x - y).count_ops(visual=True)
SUB
>>> (-x).count_ops(visual=True)
NEG
```

Here, there are two Adds and a Pow:

```
>>> (1 + a + b**2).count_ops(visual=True)
2*ADD + POW
```

In the following, an Add, Mul, Pow and two functions:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=True)
ADD + MUL + POW + 2*SIN
```

for a total of 5:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=False)
5
```

Note that “what you type” is not always what you get. The expression `1/x/y` is translated by sympy into `1/(x*y)` so it gives a DIV and MUL rather than two DIVs:

```
>>> (1/x/y).count_ops(visual=True)
DIV + MUL
```

The visual option can be used to demonstrate the difference in operations for expressions in different forms. Here, the Horner representation is compared with the expanded form of a polynomial:

```
>>> eq=x*(1 + x*(2 + x*(3 + x)))
>>> count_ops(eq.expand(), visual=True) - count_ops(eq, visual=True)
-MUL + 3*POW
```

The count_ops function also handles iterables:

```
>>> count_ops([x, sin(x), None, True, x + 2], visual=False)
2
>>> count_ops([x, sin(x), None, True, x + 2], visual=True)
ADD + SIN
>>> count_ops({x: sin(x), x + 2: y + 1}, visual=True)
2*ADD + SIN
```

expand_mul

`sympy.core.function.expand_mul(expr, deep=True)`

Wrapper around `expand` that only uses the `mul` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import symbols, expand_mul, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_mul(exp(x+y)*(x+y)*log(x*y**2))
x*exp(x + y)*log(x*y**2) + y*exp(x + y)*log(x*y**2)
```

expand_log

`sympy.core.function.expand_log(expr, deep=True, force=False)`

Wrapper around `expand` that only uses the `log` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import symbols, expand_log, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_log(exp(x+y)*(x+y)*log(x*y**2))
(x + y)*(log(x) + 2*log(y))*exp(x + y)
```

expand_func

`sympy.core.function.expand_func(expr, deep=True)`

Wrapper around `expand` that only uses the `func` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_func, gamma
>>> from sympy.abc import x
>>> expand_func(gamma(x + 2))
x*(x + 1)*gamma(x)
```

expand_trig

`sympy.core.function.expand_trig(expr, deep=True)`

Wrapper around `expand` that only uses the `trig` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_trig, sin
>>> from sympy.abc import x, y
>>> expand_trig(sin(x+y)*(x+y))
(x + y)*(sin(x)*cos(y) + sin(y)*cos(x))
```

expand_complex

`sympy.core.function.expand_complex(expr, deep=True)`

Wrapper around `expand` that only uses the `complex` hint. See the `expand` docstring for more information.

See also:

`Expr.as_real_imag`

Examples

```
>>> from sympy import expand_complex, exp, sqrt, I
>>> from sympy.abc import z
>>> expand_complex(exp(z))
I*exp(re(z))*sin(im(z)) + exp(re(z))*cos(im(z))
>>> expand_complex(sqrt(I))
sqrt(2)/2 + sqrt(2)*I/2
```

expand_multinomial

`sympy.core.function.expand_multinomial(expr, deep=True)`

Wrapper around `expand` that only uses the `multinomial` hint. See the `expand` docstring for more information.

Examples

```
>>> from sympy import symbols, expand_multinomial, exp
>>> x, y = symbols('x y', positive=True)
>>> expand_multinomial((x + exp(x + 1))**2)
x**2 + 2*x*exp(x + 1) + exp(2*x + 2)
```

expand_power_exp

`sympy.core.function.expand_power_exp(expr, deep=True)`
 Wrapper around `expand` that only uses the `power_exp` hint.
 See the `expand` docstring for more information.

Examples

```
>>> from sympy import expand_power_exp
>>> from sympy.abc import x, y
>>> expand_power_exp(x**(y + 2))
x**2*x**y
```

expand_power_base

`sympy.core.function.expand_power_base(expr, deep=True, force=False)`
 Wrapper around `expand` that only uses the `power_base` hint.
 See the `expand` docstring for more information.

A wrapper to `expand(power_base=True)` which separates a power with a base that is a `Mul` into a product of powers, without performing any other expansions, provided that assumptions about the power's base and exponent allow.

`deep=False` (default is `True`) will only apply to the top-level expression.

`force=True` (default is `False`) will cause the expansion to ignore assumptions about the base and exponent. When `False`, the expansion will only happen if the base is non-negative or the exponent is an integer.

```
>>> from sympy.abc import x, y, z
>>> from sympy import expand_power_base, sin, cos, exp
```

```
>>> (x*y)**2
x**2*y**2
```

```
>>> (2*x)**y
(2*x)**y
>>> expand_power_base(_)
2**y*x**y
```

```
>>> expand_power_base((x*y)**z)
(x*y)**z
>>> expand_power_base((x*y)**z, force=True)
```

(continues on next page)

(continued from previous page)

```
x**z*y**z
>>> expand_power_base(sin((x*y)**z), deep=False)
sin((x*y)**z)
>>> expand_power_base(sin((x*y)**z), force=True)
sin(x**z*y**z)
```

```
>>> expand_power_base((2*sin(x))**y + (2*cos(x))**y)
2**y*sin(x)**y + 2**y*cos(x)**y
```

```
>>> expand_power_base((2*exp(y))**x)
2**x*exp(y)**x
```

```
>>> expand_power_base((2*cos(x))**y)
2**y*cos(x)**y
```

Notice that sums are left untouched. If this is not the desired behavior, apply full `expand()` to the expression:

```
>>> expand_power_base(((x+y)*z)**2)
z**2*(x + y)**2
>>> (((x+y)*z)**2).expand()
x**2*z**2 + 2*x*y*z**2 + y**2*z**2
```

```
>>> expand_power_base((2*y)**(1+z))
2**(z + 1)*y**(z + 1)
>>> ((2*y)**(1+z)).expand()
2*2**z*y*y**z
```

nfloat

`sympy.core.function.nfloat(expr, n=15, exponent=False)`

Make all Rationals in `expr` Floats except those in exponents (unless the `exponents` flag is set to `True`).

Examples

```
>>> from sympy.core.function import nfloat
>>> from sympy.abc import x, y
>>> from sympy import cos, pi, sqrt
>>> nfloat(x**4 + x/2 + cos(pi/3) + 1 + sqrt(y))
x**4 + 0.5*x + sqrt(y) + 1.5
>>> nfloat(x**4 + sqrt(y), exponent=True)
x**4.0 + y**0.5
```

5.1.20 evalf

PrecisionExhausted

`class sympy.core.evalf.PrecisionExhausted`

N

`sympy.core.evalf.N(x, n=15, **options)`

Calls `x.evalf(n, **options)`.

Both `.n()` and `N()` are equivalent to `.evalf()`; use the one that you like better. See also the docstring of `.evalf()` for information on the options.

Examples

```
>>> from sympy import Sum, oo, N
>>> from sympy.abc import k
>>> Sum(1/k**k, (k, 1, oo))
Sum(k**(-k), (k, 1, oo))
>>> N(_, 4)
1.291
```

5.1.21 containers

Tuple

class `sympy.core.containers.Tuple`

Wrapper around the builtin tuple object

The Tuple is a subclass of Basic, so that it works well in the SymPy framework. The wrapped tuple is available as `self.args`, but you can also access elements or slices with `[:]` syntax.

Parameters `sympify` : bool

If False, `sympify` is not called on `args`. This can be used for speedups for very large tuples where the elements are known to already be sympy objects.

Example

```
>>> from sympy import symbols
>>> from sympy.core.containers import Tuple
>>> a, b, c, d = symbols('a b c d')
>>> Tuple(a, b, c)[1:]
(b, c)
>>> Tuple(a, b, c).subs(a, d)
(d, b, c)
```

index(`value`[, `start`[, `stop`]]) → integer - return first index of value.

Raises `ValueError` if the value is not present.

tuple_count(`value`)

`T.count(value)` -> integer - return number of occurrences of value

Dict

`class sympy.core.containers.Dict`

Wrapper around the builtin dict object

The Dict is a subclass of Basic, so that it works well in the SymPy framework. Because it is immutable, it may be included in sets, but its values must all be given at instantiation and cannot be changed afterwards. Otherwise it behaves identically to the Python dict.

```
>>> from sympy.core.containers import Dict
```

```
>>> D = Dict({1: 'one', 2: 'two'})
>>> for key in D:
...     if key == 1:
...         print('%s %s' % (key, D[key]))
1 one
```

The args are sympified so the 1 and 2 are Integers and the values are Symbols. Queries automatically sympify args so the following work:

```
>>> 1 in D
True
>>> D.has('one') # searches keys and values
True
>>> 'one' in D # not in the keys
False
>>> D[1]
one
```

get(*k*, *d*) → *D*[*k*] if *k* in *D*, else *d*. *d* defaults to None.

items() → list of *D*'s (key, value) pairs, as 2-tuples

keys() → list of *D*'s keys

values() → list of *D*'s values

5.1.22 compatibility

iterable

`sympy.core.compatibility.iterable(i, exclude=((<class 'str'>, <class 'dict'>, <class 'sympy.core.compatibility.NotIterable'>)))`

Return a boolean indicating whether *i* is SymPy iterable. True also indicates that the iterator is finite, i.e. you e.g. call `list(...)` on the instance.

When SymPy is working with iterables, it is almost always assuming that the iterable is not a string or a mapping, so those are excluded by default. If you want a pure Python definition, make `exclude=None`. To exclude multiple items, pass them as a tuple.

You can also set the `_iterable` attribute to True or False on your class, which will override the checks here, including the exclude test.

As a rule of thumb, some SymPy functions use this to check if they should recursively map over an object. If an object is technically iterable in the Python sense but does not desire this behavior (e.g., because its iteration is not finite, or because iteration might

induce an unwanted computation), it should disable it by setting the `_iterable` attribute to `False`.

See also: `is_sequence`

Examples

```
>>> from sympy.utilities.iterables import iterable
>>> from sympy import Tuple
>>> things = [[1], (1,), set([1]), Tuple(1), (j for j in [1, 2]), {1:2}, '1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'sympy.core.containers.Tuple'>
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>
```

```
>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```

`is_sequence`

`sympy.core.compatibility.is_sequence(i, include=None)`

Return a boolean indicating whether `i` is a sequence in the SymPy sense. If anything that fails the test below should be included as being a sequence for your application, set `'include'` to that object's type; multiple types should be passed as a tuple of types.

Note: although generators can generate a sequence, they often need special handling to make sure their elements are captured before the generator is exhausted, so these are not included by default in the definition of a sequence.

See also: `iterable`

Examples

```
>>> from sympy.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
```

(continues on next page)

(continued from previous page)

```
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

as_int

`sympy.core.compatibility.as_int(n)`

Convert the argument to a builtin integer.

The return value is guaranteed to be equal to the input. `ValueError` is raised if the input has a non-integral value.

Examples

```
>>> from sympy.core.compatibility import as_int
>>> from sympy import sqrt
>>> 3.0
3.0
>>> as_int(3.0) # convert to int and test for equality
3
>>> int(sqrt(10))
3
>>> as_int(sqrt(10))
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

5.1.23 exprtools

gcd_terms

`sympy.core.exprtools.gcd_terms(terms, isprimitive=False, clear=True, fraction=True)`

Compute the GCD of terms and put them together.

`terms` can be an expression or a non-Basic sequence of expressions which will be handled as though they are terms from a sum.

If `isprimitive` is `True` the `_gcd_terms` will not run the primitive method on the terms.

`clear` controls the removal of integers from the denominator of an Add expression. When `True` (default), all numerical denominator will be cleared; when `False` the denominators will be cleared only if all terms had numerical denominators other than 1.

`fraction`, when `True` (default), will put the expression over a common denominator.

See also:

`factor_terms` (page 201), `sympy.polys.polytools.terms_gcd` (page 811)

Examples

```
>>> from sympy.core import gcd_terms
>>> from sympy.abc import x, y
```

```
>>> gcd_terms((x + 1)**2*y + (x + 1)*y**2)
y*(x + 1)*(x + y + 1)
>>> gcd_terms(x/2 + 1)
(x + 2)/2
>>> gcd_terms(x/2 + 1, clear=False)
x/2 + 1
>>> gcd_terms(x/2 + y/2, clear=False)
(x + y)/2
>>> gcd_terms(x/2 + 1/x)
(x**2 + 2)/(2*x)
>>> gcd_terms(x/2 + 1/x, fraction=False)
(x + 2/x)/2
>>> gcd_terms(x/2 + 1/x, fraction=False, clear=False)
x/2 + 1/x
```

```
>>> gcd_terms(x/2/y + 1/x/y)
(x**2 + 2)/(2*x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False)
(x**2/2 + 1)/(x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False, fraction=False)
(x/2 + 1/x)/y
```

The `clear` flag was ignored in this case because the returned expression was a rational expression, not a simple sum.

factor_terms

`sympy.core.exprtools.factor_terms(expr, radical=False, clear=False, fraction=False, sign=True)`

Remove common factors from terms in all arguments without changing the underlying structure of the `expr`. No expansion or simplification (and no processing of non-commutatives) is performed.

If `radical=True` then a radical common to all terms will be factored out of any Add sub-expressions of the `expr`.

If `clear=False` (default) then coefficients will not be separated from a single Add if they can be distributed to leave one or more terms with integer coefficients.

If `fraction=True` (default is False) then a common denominator will be constructed for the expression.

If `sign=True` (default) then even if the only factor in common is a -1, it will be factored out of the expression.

See also:

[gcd_terms](#) (page 200), [sympy.polys.polytools.terms_gcd](#) (page 811)

Examples

```
>>> from sympy import factor_terms, Symbol
>>> from sympy.abc import x, y
>>> factor_terms(x + x*(2 + 4*y)**3)
x*(8*(2*y + 1)**3 + 1)
>>> A = Symbol('A', commutative=False)
>>> factor_terms(x*A + x*A + x*y*A)
x*(y*A + 2*A)
```

When `clear` is `False`, a rational will only be factored out of an `Add` expression if all terms of the `Add` have coefficients that are fractions:

```
>>> factor_terms(x/2 + 1, clear=False)
x/2 + 1
>>> factor_terms(x/2 + 1, clear=True)
(x + 2)/2
```

If a `-1` is all that can be factored out, to not factor it out, the flag `sign` must be `False`:

```
>>> factor_terms(-x - y)
-(x + y)
>>> factor_terms(-x - y, sign=False)
-x - y
>>> factor_terms(-2*x - 2*y, sign=False)
-2*(x + y)
```

5.2 Combinatorics Module

5.2.1 Contents

Partitions

class `sympy.combinatorics.partitions.Partition`

This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.

See also:

[sympy.utilities.iterables.partitions](#) (page 1429), [sympy.utilities.iterables.multiset_partitions](#) (page 1425)

Attributes

<code>is_Complement</code>	
<code>is_EmptySet</code>	
<code>is_Intersection</code>	
<code>is_UniversalSet</code>	

RGS

Returns the “restricted growth string” of the partition.

The RGS is returned as a list of indices, L , where $L[i]$ indicates the block in which element i appears. For example, in a partition of 3 elements (a, b, c) into 2 blocks ($\{c\}, \{a, b\}$) the RGS is $[1, 1, 0]$: “ a ” is in block 1, “ b ” is in block 1 and “ c ” is in block 0.

Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
{{3}, {4}, {5}, {1, 2}}
>>> _.RGS
(0, 0, 1, 2, 3)
```

classmethod `from_rgs(rgs, elements)`

Creates a set partition from a restricted growth string.

The indices given in `rgs` are assumed to be the index of the element as given in `elements` as provided (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in `rgs` an error will be raised.

Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
{{c}, {a, d}, {b, e}}
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
{{e}, {a, c}, {b, d}}
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
{{2}, {1, 4}, {3, 5}}
```

partition

Return partition as a sorted list of lists.

Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> Partition([1], [2, 3]).partition
[[1], [2, 3]]
```

rank

Gets the rank of a partition.

Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.rank
13
```

sort_key(order=None)

Return a canonical key that can be used for sorting.

Ordering is based on the size and sorted elements of the partition and ties are broken with the rank.

Examples

```
>>> from sympy.utilities.iterables import default_sort_key
>>> from sympy.combinatorics.partitions import Partition
>>> from sympy.abc import x
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key); l
[{{1, 2}}, {{1}, {2}}, {{1, x}}, {{3, 4}}, {{0, 1, 2, 3}}]
```

class sympy.combinatorics.partitions.IntegerPartition

This class represents an integer partition.

In number theory and combinatorics, a partition of a positive integer, n , also called an integer partition, is a way of writing n as a list of positive integers that sum to n . Two partitions that differ only in the order of summands are considered to be the same partition; if order matters then the partitions are referred to as compositions. For example, 4 has five partitions: [4], [3, 1], [2, 2], [2, 1, 1], and [1, 1, 1, 1]; the compositions [1, 2, 1] and [1, 1, 2] are the same as partition [2, 1, 1].

See also:

[sympy.utilities.iterables.partitions](#) (page 1429), [sympy.utilities.iterables.multiset_partitions](#) (page 1425)

Reference http://en.wikipedia.org/wiki/Partition_%28number_theory%29

as_dict()

Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> IntegerPartition([1]*3 + [2] + [3]*4).as_dict()
{1: 3, 2: 1, 3: 4}
```

as_ferrers(char='#')

Prints the ferrer diagram of a partition.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
#####
#
#
```

conjugate

Computes the conjugate partition of itself.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> a = IntegerPartition([6, 3, 3, 2, 1])
>>> a.conjugate
[5, 4, 3, 1, 1, 1]
```

next_lex()

Return the next partition of the integer, n , in lexical order, wrapping around to $[n]$ if the partition is $[1, \dots, 1]$.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([3, 1])
>>> print(p.next_lex())
[4]
>>> p.partition < p.next_lex().partition
True
```

prev_lex()

Return the previous partition of the integer, n , in lexical order, wrapping around to $[1, \dots, 1]$ if the partition is $[n]$.

Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([4])
>>> print(p.prev_lex())
[3, 1]
>>> p.partition > p.prev_lex().partition
True
```

`sympy.combinatorics.partitions.random_integer_partition( $n$ , seed=None)`
Generates a random integer partition summing to n as a list of reverse-sorted integers.

Examples

```
>>> from sympy.combinatorics.partitions import random_integer_partition
```

For the following, a seed is given so a known value can be shown; in practice, the seed would not be given.

```
>>> random_integer_partition(100, seed=[1, 1, 12, 1, 2, 1, 85, 1])
[85, 12, 2, 1]
>>> random_integer_partition(10, seed=[1, 2, 3, 1, 5, 1])
[5, 3, 1, 1]
>>> random_integer_partition(1)
[1]
```

`sympy.combinatorics.partitions.RGS_generalized(m)`

Computes the $m + 1$ generalized unrestricted growth strings and returns them as rows in matrix.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_generalized
>>> RGS_generalized(6)
Matrix([
[ 1,  1,  1,  1,  1, 1, 1],
[ 1,  2,  3,  4,  5, 6, 0],
[ 2,  5, 10, 17, 26, 0, 0],
[ 5, 15, 37, 77, 0, 0, 0],
[15, 52, 151, 0, 0, 0, 0],
[52, 203, 0, 0, 0, 0, 0],
[203, 0, 0, 0, 0, 0, 0]])
```

`sympy.combinatorics.partitions.RGS_enum(m)`

`RGS_enum` computes the total number of restricted growth strings possible for a super-set of size m .

Examples

```
>>> from sympy.combinatorics.partitions import RGS_enum
>>> from sympy.combinatorics.partitions import Partition
>>> RGS_enum(4)
15
>>> RGS_enum(5)
52
>>> RGS_enum(6)
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```
>>> a = Partition(list(range(4)))
>>> s = set()
>>> for i in range(20):
...     s.add(a)
```

(continues on next page)

(continued from previous page)

```
...     a += 1
...
>>> assert len(s) == 15
```

`sympy.combinatorics.partitions.RGS_unrank(rank, m)`
 Gives the unranked restricted growth string for a given superset size.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_unrank
>>> RGS_unrank(14, 4)
[0, 1, 2, 3]
>>> RGS_unrank(0, 4)
[0, 0, 0, 0]
```

`sympy.combinatorics.partitions.RGS_rank(rgs)`
 Computes the rank of a restricted growth string.

Examples

```
>>> from sympy.combinatorics.partitions import RGS_rank, RGS_unrank
>>> RGS_rank([0, 1, 2, 1, 3])
42
>>> RGS_rank(RGS_unrank(4, 7))
4
```

Permutations

`class sympy.combinatorics.permutations.Permutation`

A permutation, alternatively known as an ‘arrangement number’ or ‘ordering’ is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement [R30] (page 1855). For example, if one started with elements [x, y, a, b] (in that order) and they were reordered as [x, y, b, a] then the permutation would be [0, 1, 3, 2]. Notice that (in SymPy) the first element is always referred to as 0 and the permutation uses the indices of the elements in the original ordering, not the elements (a, b, etc...) themselves.

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = False
```

See also:

[Cycle](#) (page 230)

References

[R29] (page 1855), [R30] (page 1855), [R31] (page 1855), [R32] (page 1855), [R33] (page 1855), [R34] (page 1855), [R35] (page 1855)

Permutations Notation

Permutations are commonly represented in disjoint cycle or array forms.

Array Notation And 2-line Form

In the 2-line form, the elements and their final positions are shown as a matrix with 2 rows:

$[0\ 1\ 2\ \dots\ n-1]\ [p(0)\ p(1)\ p(2)\ \dots\ p(n-1)]$

Since the first line is always $\text{range}(n)$, where n is the size of p , it is sufficient to represent the permutation by the second line, referred to as the “array form” of the permutation. This is entered in brackets as the argument to the `Permutation` class:

```
>>> p = Permutation([0, 2, 1]); p
Permutation([0, 2, 1])
```

Given i in $\text{range}(p.\text{size})$, the permutation maps i to i^p

```
>>> [i^p for i in range(p.size)]
[0, 2, 1]
```

The composite of two permutations $p*q$ means first apply p , then q , so $i^{(p*q)} = (i^p)^q$ which is $i^p{}^q$ according to Python precedence rules:

```
>>> q = Permutation([2, 1, 0])
>>> [i^p^q for i in range(3)]
[2, 0, 1]
>>> [i^(p*q) for i in range(3)]
[2, 0, 1]
```

One can use also the notation $p(i) = i^p$, but then the composition rule is $(p*q)(i) = q(p(i))$, not $p(q(i))$:

```
>>> [(p*q)(i) for i in range(p.size)]
[2, 0, 1]
>>> [q(p(i)) for i in range(p.size)]
[2, 0, 1]
>>> [p(q(i)) for i in range(p.size)]
[1, 2, 0]
```

Disjoint Cycle Notation

In disjoint cycle notation, only the elements that have shifted are indicated. In the above case, the 2 and 1 switched places. This can be entered in two ways:

```
>>> Permutation(1, 2) == Permutation([[1, 2]]) == p
True
```

Only the relative ordering of elements in a cycle matter:

```
>>> Permutation(1,2,3) == Permutation(2,3,1) == Permutation(3,1,2)
True
```

The disjoint cycle notation is convenient when representing permutations that have several cycles in them:

```
>>> Permutation(1, 2)(3, 5) == Permutation([[1, 2], [3, 5]])
True
```

It also provides some economy in entry when computing products of permutations that are written in disjoint cycle notation:

```
>>> Permutation(1, 2)(1, 3)(2, 3)
Permutation([0, 3, 2, 1])
>>> _ == Permutation([[1, 2]])*Permutation([[1, 3]])*Permutation([[2, 3]])
True
```

Caution: when the cycles have common elements between them then the order in which the permutations are applied matters. The convention is that the permutations are applied from right to left. In the following, the transposition of elements 2 and 3 is followed by the transposition of elements 1 and 2:

```
>>> Permutation(1, 2)(2, 3) == Permutation([(1, 2), (2, 3)])
True
>>> Permutation(1, 2)(2, 3).list()
[0, 3, 1, 2]
```

If the first and second elements had been swapped first, followed by the swapping of the second and third, the result would have been [0, 2, 3, 1]. If, for some reason, you want to apply the cycles in the order they are entered, you can simply reverse the order of cycles:

```
>>> Permutation([(1, 2), (2, 3)][::-1]).list()
[0, 2, 3, 1]
```

Entering a singleton in a permutation is a way to indicate the size of the permutation. The size keyword can also be used.

Array-form entry:

```
>>> Permutation([[1, 2], [9]])
Permutation([0, 2, 1], size=10)
>>> Permutation([[1, 2]], size=10)
Permutation([0, 2, 1], size=10)
```

Cyclic-form entry:

```
>>> Permutation(1, 2, size=10)
Permutation([0, 2, 1], size=10)
>>> Permutation(9)(1, 2)
Permutation([0, 2, 1], size=10)
```

Caution: no singleton containing an element larger than the largest in any previous cycle can be entered. This is an important difference in how `Permutation` and `Cycle` handle the `__call__` syntax. A singleton argument at the start of a `Permutation` performs instantiation of the `Permutation` and is permitted:

```
>>> Permutation(5)
Permutation([], size=6)
```

A singleton entered after instantiation is a call to the permutation – a function call – and if the argument is out of range it will trigger an error. For this reason, it is better to start the cycle with the singleton:

The following fails because there is no element 3:

```
>>> Permutation(1, 2)(3)
Traceback (most recent call last):
...
IndexError: list index out of range
```

This is ok: only the call to an out of range singleton is prohibited; otherwise the permutation autosizes:

```
>>> Permutation(3)(1, 2)
Permutation([0, 2, 1, 3])
>>> Permutation(1, 2)(3, 4) == Permutation(3, 4)(1, 2)
True
```

Equality Testing

The array forms must be the same in order for permutations to be equal:

```
>>> Permutation([1, 0, 2, 3]) == Permutation([1, 0])
False
```

Identity Permutation

The identity permutation is a permutation in which no element is out of place. It can be entered in a variety of ways. All the following create an identity permutation of size 4:

```
>>> I = Permutation([0, 1, 2, 3])
>>> all(p == I for p in [
...     Permutation(3),
...     Permutation(range(4)),
...     Permutation([], size=4),
...     Permutation(size=4)])
True
```

Watch out for entering the range inside a set of brackets (which is cycle notation):

```
>>> I == Permutation([range(4)])
False
```

Permutation Printing

There are a few things to note about how Permutations are printed.

1) If you prefer one form (array or cycle) over another, you can set that with the `print_cyclic` flag.

```
>>> Permutation(1, 2)(4, 5)(3, 4)
Permutation([0, 2, 1, 4, 5, 3])
>>> p = _
```



```
>>> Permutation.print_cyclic = True
>>> p
(1 2)(3 4 5)
>>> Permutation.print_cyclic = False
```

2) Regardless of the setting, a list of elements in the array for cyclic form can be obtained and either of those can be copied and supplied as the argument to Permutation:

```
>>> p.array_form
[0, 2, 1, 4, 5, 3]
>>> p.cyclic_form
[[1, 2], [3, 4, 5]]
>>> Permutation(_) == p
True
```

3) Printing is economical in that as little as possible is printed while retaining all information about the size of the permutation:

```
>>> Permutation([1, 0, 2, 3])
Permutation([1, 0, 2, 3])
>>> Permutation([1, 0, 2, 3], size=20)
Permutation([1, 0], size=20)
>>> Permutation([1, 0, 2, 4, 3, 5, 6], size=20)
Permutation([1, 0, 2, 4, 3], size=20)
```

```
>>> p = Permutation([1, 0, 2, 3])
>>> Permutation.print_cyclic = True
>>> p
(3)(0 1)
>>> Permutation.print_cyclic = False
```

The 2 was not printed but it is still there as can be seen with the array_form and size methods:

```
>>> p.array_form
[1, 0, 2, 3]
>>> p.size
4
```

Short Introduction To Other Methods

The permutation can act as a bijective function, telling what element is located at a given position

```
>>> q = Permutation([5, 2, 3, 4, 1, 0])
>>> q.array_form[1] # the hard way
2
>>> q(1) # the easy way
2
>>> {i: q(i) for i in range(q.size)} # showing the bijection
{0: 5, 1: 2, 2: 3, 3: 4, 4: 1, 5: 0}
```

The full cyclic form (including singletons) can be obtained:

```
>>> p.full_cyclic_form
[[0, 1], [2], [3]]
```

Any permutation can be factored into transpositions of pairs of elements:

```
>>> Permutation([[1, 2], [3, 4, 5]]).transpositions()
[(1, 2), (3, 5), (3, 4)]
>>> Permutation.rmul(*[Permutation([ti], size=6) for ti in _]).cyclic_form
[[1, 2], [3, 4, 5]]
```

The number of permutations on a set of n elements is given by $n!$ and is called the cardinality.

```
>>> p.size
4
>>> p.cardinality
24
```

A given permutation has a rank among all the possible permutations of the same elements, but what that rank is depends on how the permutations are enumerated. (There are a number of different methods of doing so.) The lexicographic rank is given by the rank method and this rank is used to increment a permutation with addition/subtraction:

```
>>> p.rank()
6
>>> p + 1
Permutation([1, 0, 3, 2])
>>> p.next_lex()
Permutation([1, 0, 3, 2])
>>> _.rank()
7
>>> p.unrank_lex(p.size, rank=7)
Permutation([1, 0, 3, 2])
```

The product of two permutations p and q is defined as their composition as functions, $(p*q)(i) = q(p(i))$ [R34] (page 1855).

```
>>> p = Permutation([1, 0, 2, 3])
>>> q = Permutation([2, 3, 1, 0])
>>> list(q*p)
[2, 3, 0, 1]
>>> list(p*q)
[3, 2, 1, 0]
>>> [q(p(i)) for i in range(p.size)]
[3, 2, 1, 0]
```

The permutation can be ‘applied’ to any list-like object, not only Permutations:

```
>>> p(['zero', 'one', 'four', 'two'])
['one', 'zero', 'four', 'two']
>>> p('zo42')
['o', 'z', '4', '2']
```

If you have a list of arbitrary elements, the corresponding permutation can be found with the from_sequence method:

```
>>> Permutation.from_sequence('SymPy')
Permutation([1, 3, 2, 0, 4])
```

array_form

Return a copy of the attribute `_array_form` Examples =====

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([[2, 0], [3, 1]])
>>> p.array_form
[2, 3, 0, 1]
>>> Permutation([[2, 0, 3, 1]]).array_form
[3, 2, 0, 1]
>>> Permutation([2, 0, 3, 1]).array_form
[2, 0, 3, 1]
>>> Permutation([[1, 2], [4, 5]]).array_form
[0, 2, 1, 3, 5, 4]
```

ascents()

Returns the positions of ascents in a permutation, ie, the location where $p[i] < p[i+1]$

See also:

[descents](#) (page 215), [inversions](#) (page 219), [min](#) (page 223), [max](#) (page 223)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.ascents()
[1, 2]
```

atoms()

Returns all the elements of a permutation

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2, 3, 4, 5]).atoms()
{0, 1, 2, 3, 4, 5}
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
{0, 1, 2, 3, 4, 5}
```

cardinality

Returns the number of all possible permutations.

See also:

[length](#) (page 222), [order](#) (page 224), [rank](#) (page 225), [size](#) (page 228)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.cardinality
24
```

commutator(x)

Return the commutator of self and x: $\sim x \sim \text{self} * x * \text{self}$

If f and g are part of a group, G, then the commutator of f and g is the group identity iff f and g commute, i.e. $fg == gf$.

References

<http://en.wikipedia.org/wiki/Commutator>

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([0, 2, 3, 1])
>>> x = Permutation([2, 0, 3, 1])
>>> c = p.commutator(x); c
Permutation([2, 1, 3, 0])
>>> c == ~x*~p*x*p
True
```

```
>>> I = Permutation(3)
>>> p = [I + i for i in range(6)]
>>> for i in range(len(p)):
...     for j in range(len(p)):
...         c = p[i].commutator(p[j])
...         if p[i]*p[j] == p[j]*p[i]:
...             assert c == I
...         else:
...             assert c != I
... 
```

commutes_with(other)

Checks if the elements are commuting.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([1, 4, 3, 0, 2, 5])
>>> b = Permutation([0, 1, 2, 3, 4, 5])
>>> a.commutates_with(b)
True
>>> b = Permutation([2, 3, 5, 4, 1, 0])
>>> a.commutates_with(b)
False
```

cycle_structure

Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation(0, 4, 3)(1, 2)(5, 6).cycle_structure
{2: 2, 3: 1}
```

cycles

Returns the number of cycles contained in the permutation (including singletons).

See also:

[sympy.functions.combinatorial.numbers.stirling](#) (page 473)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation(0, 1)(2, 3).cycles
2
```

cyclic_form

This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

See also:

[array_form](#) (page 212), [full_cyclic_form](#) (page 216)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([0, 3, 1, 2])
>>> p.cyclic_form
[[1, 3, 2]]
>>> Permutation([1, 0, 2, 4, 3, 5]).cyclic_form
[[0, 1], [3, 4]]
```

descents()

Returns the positions of descents in a permutation, ie, the location where $p[i] > p[i+1]$

See also:

[ascents](#) (page 213), [inversions](#) (page 219), [min](#) (page 223), [max](#) (page 223)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.descents()
[0, 3]
```

classmethod from_inversion_vector(inversion)
Calculates the permutation from the inversion vector.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> Permutation.from_inversion_vector([3, 2, 1, 0, 0])
Permutation([3, 2, 1, 0, 4, 5])
```

classmethod from_sequence(i, key=None)
Return the permutation needed to obtain i from the sorted elements of i. If custom sorting is desired, a key can be given.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
```

```
>>> Permutation.from_sequence('SymPy')
(4)(0 1 3)
>>> _((sorted("SymPy")))
['S', 'y', 'm', 'P', 'y']
>>> Permutation.from_sequence('SymPy', key=lambda x: x.lower())
(4)(0 2)(1 3)
```

full_cyclic_form
Return permutation in cyclic form including singletons.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation([0, 2, 1]).full_cyclic_form
[[0], [1, 2]]
```

get_adjacency_distance(other)
Computes the adjacency distance between two permutations.
This metric counts the number of times a pair i,j of jobs is adjacent in both p and p'. If n_adj is this quantity then the adjacency distance is n - n_adj - 1 [1]
[1] Reeves, Colin R. Landscapes, Operators and Heuristic search, Annals of Operational Research, 86, pp 473-490. (1999)

See also:

[get_precedence_matrix](#) (page 218), [get_precedence_distance](#) (page 218),
[get_adjacency_matrix](#) (page 217)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> p.get_adjacency_distance(q)
3
>>> r = Permutation([0, 2, 1, 4, 3])
>>> p.get_adjacency_distance(r)
4
```

get_adjacency_matrix()

Computes the adjacency matrix of a permutation.

If job i is adjacent to job j in a permutation p then we set $m[i, j] = 1$ where m is the adjacency matrix of p .

See also:

[get_precedence_matrix](#) (page 218), [get_precedence_distance](#) (page 218),
[get_adjacency_distance](#) (page 216)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation.josephus(3, 6, 1)
>>> p.get_adjacency_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1],
[0, 1, 0, 0, 0, 0],
[1, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 0]])
>>> q = Permutation([0, 1, 2, 3])
>>> q.get_adjacency_matrix()
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[0, 0, 0, 0]])
```

get_positional_distance(other)

Computes the positional distance between two permutations.

See also:

[get_precedence_distance](#) (page 218), [get_adjacency_distance](#) (page 216)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> r = Permutation([3, 1, 4, 0, 2])
>>> p.get_positional_distance(q)
12
>>> p.get_positional_distance(r)
12
```

get_precedence_distance(other)

Computes the precedence distance between two permutations.

Suppose p and p' represent n jobs. The precedence metric counts the number of times a job j is preceded by job i in both p and p' . This metric is commutative.

See also:

[get_precedence_matrix](#) (page 218), [get_adjacency_matrix](#) (page 217),
[get_adjacency_distance](#) (page 216)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 0, 4, 3, 1])
>>> q = Permutation([3, 1, 2, 4, 0])
>>> p.get_precedence_distance(q)
7
>>> q.get_precedence_distance(p)
7
```

get_precedence_matrix()

Gets the precedence matrix. This is used for computing the distance between two permutations.

See also:

[get_precedence_distance](#) (page 218), [get_adjacency_matrix](#) (page 217),
[get_adjacency_distance](#) (page 216)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation.josephus(3, 6, 1)
>>> p
Permutation([2, 5, 3, 1, 4, 0])
>>> p.get_precedence_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[1, 0, 0, 0, 1, 0],
[1, 1, 0, 1, 1, 1],
[1, 1, 0, 0, 1, 0],
[1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 0]])
```


index()

Returns the index of a permutation.

The index of a permutation is the sum of all subscripts j such that $p[j]$ is greater than $p[j+1]$.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([3, 0, 2, 1, 4])
>>> p.index()
2
```

inversion_vector()

Return the inversion vector of the permutation.

The inversion vector consists of elements whose value indicates the number of elements in the permutation that are lesser than it and lie on its right hand side.

The inversion vector is the same as the Lehmer encoding of a permutation.

See also:

[from_inversion_vector](#) (page 216)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the i -th element cycling through $0..i$.

```
>>> p = Permutation(2)
>>> while p:
...     print('%s %s %s' % (p, p.inversion_vector(), p.rank()))
...     p = p.next_lex()
...
Permutation([0, 1, 2]) [0, 0] 0
Permutation([0, 2, 1]) [0, 1] 1
Permutation([1, 0, 2]) [1, 0] 2
Permutation([1, 2, 0]) [1, 1] 3
Permutation([2, 0, 1]) [2, 0] 4
Permutation([2, 1, 0]) [2, 1] 5
```

inversions()

Computes the number of inversions of a permutation.

An inversion is where $i > j$ but $p[i] < p[j]$.

For small length of p , it iterates over all i and j values and calculates the number of inversions. For large length of p , it uses a variation of merge sort to calculate the number of inversions.

See also:

[descents](#) (page 215), [ascents](#) (page 213), [min](#) (page 223), [max](#) (page 223)

References

[1] <http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3, 4, 5])
>>> p.inversions()
0
>>> Permutation([3, 2, 1, 0]).inversions()
6
```

is_Empty

Checks to see if the permutation is a set with zero elements

See also:

[is_Singleton](#) (page 221)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

is_Identity

Returns True if the Permutation is an identity permutation.

See also:

[order](#) (page 224)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([])
>>> p.is_Identity
True
>>> p = Permutation([[0], [1], [2]])
>>> p.is_Identity
True
>>> p = Permutation([0, 1, 2])
>>> p.is_Identity
True
>>> p = Permutation([0, 2, 1])
```

(continues on next page)

(continued from previous page)

```
>>> p.is_Identity
False
```

is_Singleton

Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

See also:

[is_Empty](#) (page 220)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False
```

is_even

Checks if a permutation is even.

See also:

[is_odd](#) (page 221)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_even
True
>>> p = Permutation([3, 2, 1, 0])
>>> p.is_even
True
```

is_odd

Checks if a permutation is odd.

See also:

[is_even](#) (page 221)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_odd
False
>>> p = Permutation([3, 2, 0, 1])
>>> p.is_odd
True
```

classmethod josephus(m, n, s=1)

Return as a permutation the shuffling of range(n) using the Josephus scheme in which every m-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter s stops the selection process when there are s items remaining and these are selected by continuing the selection, counting by 1 rather than by m.

Consider selecting every 3rd item from 6 until only 2 remain:

choices	chosen
=====	=====
012345	
01 345	2
01 34	25
01 4	253
0 4	2531
0	25314
	253140

References

1. http://en.wikipedia.org/wiki/Flavius_Josephus
2. http://en.wikipedia.org/wiki/Josephus_problem
3. <http://www.wou.edu/~burtonl/josephus.html>

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]
```

length()

Returns the number of integers moved by a permutation.

See also:

[min](#) (page 223), [max](#) (page 223), [support](#) (page 228), [cardinality](#) (page 213), [order](#) (page 224), [rank](#) (page 225), [size](#) (page 228)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4
```

list(size=None)

Return the permutation as an explicit list, possibly trimming unremoved elements if size is less than the maximum element in the permutation; if this is desired, setting size=-1 will guarantee such trimming.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Permutation(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
>>> Permutation(3).list(-1)
[]
```

max()

The maximum element moved by the permutation.

See also:

[min](#) (page 223), [descents](#) (page 215), [ascents](#) (page 213), [inversions](#) (page 219)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([1, 0, 2, 3, 4])
>>> p.max()
1
```

min()

The minimum element moved by the permutation.

See also:

[max](#) (page 223), [descents](#) (page 215), [ascents](#) (page 213), [inversions](#) (page 219)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 4, 3, 2])
>>> p.min()
2
```

mul_inv(other)

other*~self, self and other have `_array_form`

next_lex()

Returns the next permutation in lexicographical order. If self is the last permutation in lexicographical order it returns None. See [4] section 2.4.

See also:

[rank](#) (page 225), [unrank_lex](#) (page 229)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0]); p.rank()
17
>>> p = p.next_lex(); p.rank()
18
```

next_nonlex()

Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.

See also:

[rank_nonlex](#) (page 226), [unrank_nonlex](#) (page 229)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([2, 0, 3, 1]); p.rank_nonlex()
5
>>> p = p.next_nonlex(); p
Permutation([3, 0, 1, 2])
>>> p.rank_nonlex()
6
```

next_trotterjohnson()

Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4. If it is desired to generate all such permutations, they can be generated in order more quickly with the `generate_bell` function.

See also:

[rank_trotterjohnson](#) (page 226), [unrank_trotterjohnson](#) (page 229), [sympy.utilities.iterables.generate_bell](#) (page 1419)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 0, 2, 1])
>>> p.rank_trotterjohnson()
4
>>> p = p.next_trotterjohnson(); p
Permutation([0, 3, 2, 1])
>>> p.rank_trotterjohnson()
5
```

order()

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

See also:

`identity`, `cardinality` (page 213), `length` (page 222), `rank` (page 225), `size` (page 228)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

parity()

Computes the parity of a permutation.

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that $x > y$ but $p[x] < p[y]$.

See also:

`_af_parity`

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.parity()
0
>>> p = Permutation([3, 2, 0, 1])
>>> p.parity()
1
```

classmethod random(n)

Generates a random permutation of length n .

Uses the underlying Python pseudo-random number generator.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0, 1]))
True
```

rank()

Returns the lexicographic rank of the permutation.

See also:

`next_lex` (page 223), `unrank_lex` (page 229), `cardinality` (page 213), `length` (page 222), `order` (page 224), `size` (page 228)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

rank_nonlex(inv_perm=None)

This is a linear time ranking algorithm that does not enforce lexicographic order [3].

See also:

[next_nonlex](#) (page 224), [unrank_nonlex](#) (page 229)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_nonlex()
23
```

rank_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

See also:

[unrank_trotterjohnson](#) (page 229), [next_trotterjohnson](#) (page 224)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0, 2, 1, 3])
>>> p.rank_trotterjohnson()
7
```

static **rmul**(*args)

Return product of Permutations [a, b, c, ...] as the Permutation whose ith value is a(b(c(i))).

a, b, c, ... can be Permutation objects or tuples.

Notes

All items in the sequence will be parsed by Permutation as necessary as long as the first item is a Permutation:


```
>>> Permutation.rmul(a, [0, 2, 1]) == Permutation.rmul(a, b)
True
```

The reverse order of arguments will raise a `TypeError`.

Examples

```
>>> from sympy.combinatorics.permutations import _af_rmul, Permutation
>>> Permutation.print_cyclic = False
```

```
>>> a, b = [1, 0, 2], [0, 2, 1]
>>> a = Permutation(a); b = Permutation(b)
>>> list(Permutation.rmul(a, b))
[1, 2, 0]
>>> [a(b(i)) for i in range(3)]
[1, 2, 0]
```

This handles the operands in reverse order compared to the `*` operator:

```
>>> a = Permutation(a); b = Permutation(b)
>>> list(a*b)
[2, 0, 1]
>>> [b(a(i)) for i in range(3)]
[2, 0, 1]
```

classmethod `rmul_with_af(*args)`

same as `rmul`, but the elements of `args` are `Permutation` objects which have `_array_form`

runs()

Returns the runs of a permutation.

An ascending sequence in a permutation is called a run [5].

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 5, 7, 3, 6, 0, 1, 4, 8])
>>> p.runs()
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1, 3, 2, 0])
>>> q.runs()
[[1, 3], [2], [0]]
```

signature()

Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

The signature is calculated as $(-1)^{\text{number of inversions}}$

See also:

[inversions](#) (page 219)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2])
>>> p.inversions()
0
>>> p.signature()
1
>>> q = Permutation([0, 2, 1])
>>> q.inversions()
1
>>> q.signature()
-1
```

size

Returns the number of elements in the permutation.

See also:

[cardinality](#) (page 213), [length](#) (page 222), [order](#) (page 224), [rank](#) (page 225)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([[3, 2], [0, 1]]).size
4
```

support()

Return the elements in permutation, P, for which $P[i] \neq i$.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[3, 2], [0, 1], [4]])
>>> p.array_form
[1, 0, 3, 2, 4]
>>> p.support()
[0, 1, 2, 3]
```

transpositions()

Return the permutation decomposed into a list of transpositions.

It is always possible to express a permutation as the product of transpositions, see [1]

References

1. http://en.wikipedia.org/wiki/Transposition_%28mathematics%29#Properties

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([[1, 2, 3], [0, 4, 5, 6, 7]])
>>> t = p.transpositions()
>>> t
[(0, 7), (0, 6), (0, 5), (0, 4), (1, 3), (1, 2)]
>>> print(''.join(str(c) for c in t))
(0, 7)(0, 6)(0, 5)(0, 4)(1, 3)(1, 2)
>>> Permutation.rmul(*[Permutation([ti], size=p.size) for ti in t]) == p
True
```

classmethod unrank_lex(size, rank)
Lexicographic permutation unranking.

See also:

[rank](#) (page 225), [next_lex](#) (page 223)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
>>> a
Permutation([0, 2, 4, 1, 3])
```

classmethod unrank_nonlex(n, r)
This is a linear time unranking algorithm that does not respect lexicographic order [3].

See also:

[next_nonlex](#) (page 224), [rank_nonlex](#) (page 226)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

classmethod unrank_trotterjohnson(size, rank)
Trotter Johnson permutation unranking. See [4] section 2.4.

See also:

[rank_trotterjohnson](#) (page 226), [next_trotterjohnson](#) (page 224)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

class sympy.combinatorics.permutations.Cycle(*args)

Wrapper around dict which provides the functionality of a disjoint cycle.

A cycle shows the rule to use to move subsets of elements to obtain a permutation. The Cycle class is more flexible than Permutation in that 1) all elements need not be present in order to investigate how multiple cycles act in sequence and 2) it can contain singletons:

```
>>> from sympy.combinatorics.permutations import Perm, Cycle
```

A Cycle will automatically parse a cycle given as a tuple on the rhs:

```
>>> Cycle(1, 2)(2, 3)
(1 3 2)
```

The identity cycle, Cycle(), can be used to start a product:

```
>>> Cycle()(1, 2)(2, 3)
(1 3 2)
```

The array form of a Cycle can be obtained by calling the list method (or passing it to the list function) and all elements from 0 will be shown:

```
>>> a = Cycle(1, 2)
>>> a.list()
[0, 2, 1]
>>> list(a)
[0, 2, 1]
```

If a larger (or smaller) range is desired use the list method and provide the desired size – but the Cycle cannot be truncated to a size smaller than the largest element that is out of place:

```
>>> b = Cycle(2, 4)(1, 2)(3, 1, 4)(1, 3)
>>> b.list()
[0, 2, 1, 3, 4]
>>> b.list(b.size + 1)
[0, 2, 1, 3, 4, 5]
>>> b.list(-1)
[0, 2, 1]
```

Singletons are not shown when printing with one exception: the largest element is always shown – as a singleton if necessary:

```
>>> Cycle(1, 4, 10)(4, 5)
(1 5 4 10)
>>> Cycle(1, 2)(4)(5)(10)
(1 2)(10)
```

The array form can be used to instantiate a Permutation so other properties of the permutation can be investigated:

```
>>> Perm(Cycle(1, 2)(3, 4).list()).transpositions()
[(1, 2), (3, 4)]
```

See also:

[Permutation](#) (page 207)

Notes

The underlying structure of the Cycle is a dictionary and although the `__iter__` method has been redefined to give the array form of the cycle, the underlying dictionary items are still available with the such methods as `items()`:

```
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

list(size=None)

Return the cycles as an explicit list starting from 0 up to the greater of the largest value in the cycles and size.

Truncation of trailing unmoved items will occur when size is less than the maximum element in the cycle; if this is desired, setting `size=-1` will guarantee such trimming.

Examples

```
>>> from sympy.combinatorics.permutations import Cycle
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Cycle(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Cycle(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
```

Generators**generators.symmetric()**

Generates the symmetric group of order n , S_n .

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import symmetric
>>> list(symmetric(3))
[(2), (1 2), (2)(0 1), (0 1 2), (0 2 1), (0 2)]
```

`generators.cyclic()`

Generates the cyclic group of order n , C_n .

See also:

[dihedral](#) (page 232)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import cyclic
>>> list(cyclic(5))
[(4), (0 1 2 3 4), (0 2 4 1 3),
 (0 3 1 4 2), (0 4 3 2 1)]
```

`generators.alternating()`

Generates the alternating group of order n , A_n .

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import alternating
>>> list(alternating(3))
[(2), (0 1 2), (0 2 1)]
```

`generators.dihedral()`

Generates the dihedral group of order $2n$, D_n .

The result is given as a subgroup of S_n , except for the special cases $n=1$ (the group S_2) and $n=2$ (the Klein 4-group) where that's not possible and embeddings in S_2 and S_4 respectively are given.

See also:

[cyclic](#) (page 231)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import dihedral
>>> list(dihedral(3))
[(2), (0 2), (0 1 2), (1 2), (0 2 1), (2)(0 1)]
```

Permutation Groups

`class sympy.combinatorics.perm_groups.PermutationGroup`

The class defining a Permutation group.

`PermutationGroup([p1, p2, ..., pn])` returns the permutation group generated by the list of permutations. This group can be supplied to `Polyhedron` if one desires to decorate the elements to which the indices of the permutation refer.

See also:

`sympy.combinatorics.polyhedron.Polyhedron` (page 261), `sympy.combinatorics.permutations.Permutation` (page 207)

References

- [1] Holt, D., Eick, B., O'Brien, E. "Handbook of Computational Group Theory"
- [2] Seress, A. "Permutation Group Algorithms"
- [3] http://en.wikipedia.org/wiki/Schreier_vector
- [4] http://en.wikipedia.org/wiki/Nielsen_transformation #Product_replacement_algorithm
- [5] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E.A. O'Brien. "Generating Random Elements of a Finite Group"
- [6] http://en.wikipedia.org/wiki/Block_permutation_group_theory
- [7] http://www.algorithmist.com/index.php/Union_Find
- [8] http://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups
- [9] http://en.wikipedia.org/wiki/Center_group_theory
- [10] http://en.wikipedia.org/wiki/Centralizer_and_normalizer
- [11] http://groupprops.subwiki.org/wiki/Derived_subgroup
- [12] http://en.wikipedia.org/wiki/Nilpotent_group
- [13] <http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf>

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.permutations import Cycle
>>> from sympy.combinatorics.polyhedron import Polyhedron
>>> from sympy.combinatorics.perm_groups import PermutationGroup
```

The permutations corresponding to motion of the front, right and bottom face of a 2x2 Rubik's cube are defined:

```
>>> F = Permutation(2, 19, 21, 8)(3, 17, 20, 10)(4, 6, 7, 5)
>>> R = Permutation(1, 5, 21, 14)(3, 7, 23, 12)(8, 10, 11, 9)
>>> D = Permutation(6, 18, 14, 10)(7, 19, 15, 11)(20, 22, 23, 21)
```

These are passed as permutations to `PermutationGroup`:

```
>>> G = PermutationGroup(F, R, D)
>>> G.order()
3674160
```

The group can be supplied to a `Polyhedron` in order to track the objects being moved. An example involving the 2x2 Rubik's cube is given there, but here is a simple demonstration:

```
>>> a = Permutation(2, 1)
>>> b = Permutation(1, 0)
>>> G = PermutationGroup(a, b)
>>> P = Polyhedron(list('ABC'), pgroup=G)
>>> P.corners
(A, B, C)
>>> P.rotate(0) # apply permutation 0
>>> P.corners
(A, C, B)
>>> P.reset()
>>> P.corners
(A, B, C)
```

Or one can make a permutation as a product of selected permutations and apply them to an iterable directly:

```
>>> P10 = G.make_perm([0, 1])
>>> P10('ABC')
['C', 'A', 'B']
```

base

Return a base from the Schreier-Sims algorithm.

For a permutation group G , a base is a sequence of points $B = (b_1, b_2, \dots, b_k)$ such that no element of G apart from the identity fixes all the points in B . The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

An alternative way to think of B is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.

See also:

[strong_gens](#) (page 258), [basic_transversals](#) (page 236), [basic_orbits](#) (page 235), [basic_stabilizers](#) (page 236)

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation(0, 1, 3)(2, 4)])
>>> G.base
[0, 2]
```

baseswap(base, strong_gens, pos, randomized=False, transversals=None, basic_orbits=None, strong_gens_distr=None)

Swap two consecutive base points in base and strong generating set.

If a base for a group G is given by $(b_1, b_2, \dots, b_k)$, this function returns a base $(b_1, b_2, \dots, b_{i+1}, b_i, \dots, b_k)$, where i is given by `pos`, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.

Parameters **base**, **strong_gens**

The base and strong generating set.

pos

The position at which swapping is performed.

randomized

A switch between randomized and deterministic version.

transversals

The transversals for the basic orbits, if known.

basic_orbits

The basic orbits, if known.

strong_gens_distr

The strong generators distributed by basic stabilizers, if known.

Returns (base, strong_gens)

base is the new base, and strong_gens is a generating set relative to it.

See also:

[schreier_sims](#) (page 254)

Notes

The deterministic version of the algorithm is discussed in [1], pp. 102-103; the randomized version is discussed in [1], p.103, and [2], p.98. It is of Las Vegas type. Notice that [1] contains a mistake in the pseudocode and discussion of BASESWAP: on line 3 of the pseudocode, $|\beta_{i+1}^{(T)}|$ should be replaced by $|\beta_i^{(T)}|$, and the same for the discussion of the algorithm.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> S.base
[0, 1, 2]
>>> base, gens = S.baseswap(S.base, S.strong_gens, 1, randomized=False)
>>> base, gens
([0, 2, 1],
 [(0 1 2 3), (3)(0 1), (1 3 2),
  (2 3), (1 3)])
```

check that base, gens is a BSGS

```
>>> S1 = PermutationGroup(gens)
>>> _verify_bsgs(S1, base, gens)
True
```

basic_orbits

Return the basic orbits relative to a base and strong generating set.

If $(b_1, b_2, \dots, b_k)$ is a base for a group G , and $G^{(i)} = G_{b_1, b_2, \dots, b_{i-1}}$ is the i -th basic stabilizer (so that $G^{(1)} = G$), the i -th basic orbit relative to this base is the orbit of b_i under $G^{(i)}$. See [1], pp. 87-89 for more information.

See also:

[base](#) (page 234), [strong_gens](#) (page 258), [basic_transversals](#) (page 236), [basic_stabilizers](#) (page 236)

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[0, 1, 2, 3], [1, 2, 3], [2, 3]]
```

basic_stabilizers

Return a chain of stabilizers relative to a base and strong generating set.

The i -th basic stabilizer $G^{(i)}$ relative to a base $(b_1, b_2, \dots, b_k)$ is $G_{b_1, b_2, \dots, b_{i-1}}$. For more information, see [1], pp. 87-89.

See also:

[base](#) (page 234), [strong_gens](#) (page 258), [basic_orbits](#) (page 235), [basic_transversals](#) (page 236)

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> A.base
[0, 1]
>>> for g in A.basic_stabilizers:
...     print(g)
...
PermutationGroup([
  (3)(0 1 2),
  (1 2 3)])
PermutationGroup([
  (1 2 3)])
```

basic_transversals

Return basic transversals relative to a base and strong generating set.

The basic transversals are transversals of the basic orbits. They are provided as a list of dictionaries, each dictionary having keys - the elements of one of the basic orbits, and values - the corresponding transversal elements. See [1], pp. 87-89 for more information.

See also:

[strong_gens](#) (page 258), [base](#) (page 234), [basic_orbits](#) (page 235), [basic_stabilizers](#) (page 236)

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.basic_transversals
[{0: (3), 1: (3)(0 1 2), 2: (3)(0 2 1), 3: (0 3 1)}, {1: (3), 2: (1 2 3), 3:
↪ (1 3 2)}]
```

center()

Return the center of a permutation group.

The center for a group G is defined as $Z(G) = \{z \in G | \forall g \in G, zg = gz\}$, the set of elements of G that commute with all elements of G . It is equal to the centralizer of G inside G , and is naturally a subgroup of G ([9]).

See also:

[centralizer](#) (page 237)

Notes

This is a naive implementation that is a straightforward application of [.centralizer\(\)](#)

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

centralizer(other)

Return the centralizer of a group/set/element.

The centralizer of a set of permutations S inside a group G is the set of elements of G that commute with all elements of S :

```
'C_G(S) = \{ g \in G | gs = sg \text{ for all } s \in S \}' ([10])
```

Usually, S is a subset of G , but if G is a proper subgroup of the full symmetric group, we allow for S to have elements outside G .

It is naturally a subgroup of G ; the centralizer of a permutation group is equal to the centralizer of any set of generators for that group, since any element commuting with the generators commutes with any product of the generators.

Parameters other

a permutation group/list of permutations/single permutation

See also:

[subgroup_search](#) (page 258)

Notes

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group G .

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

commutator(G, H)

Return the commutator of two subgroups.

For a permutation group K and subgroups G, H , the commutator of G and H is defined as the group generated by all the commutators $[g, h] = hgh^{-1}g^{-1}$ for g in G and h in H . It is naturally a subgroup of K ([1], p.27).

See also:

[derived_subgroup](#) (page 242)

Notes

The commutator of two subgroups H, G is equal to the normal closure of the commutators of all the generators, i.e. $hgh^{-1}g^{-1}$ for h a generator of H and g a generator of G ([1], p.28)

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

contains(g, strict=True)

Test if permutation g belong to self, G .

If g is an element of G it can be written as a product of factors drawn from the cosets of G 's stabilizers. To see if g is one of the actual generators defining the group use `G.has(g)`.

If `strict` is not `True`, g will be resized, if necessary, to match the size of permutations in self.

See also:

[coset_factor](#) (page 239), `has`, `in`

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
```

```
>>> a = Permutation(1, 2)
>>> b = Permutation(2, 3, 1)
>>> G = PermutationGroup(a, b, degree=5)
>>> G.contains(G[0]) # trivial check
True
>>> elem = Permutation([[2, 3]], size=5)
>>> G.contains(elem)
True
>>> G.contains(Permutation(4)(0, 1, 2, 3))
False
```

If strict is False, a permutation will be resized, if necessary:

```
>>> H = PermutationGroup(Permutation(5))
>>> H.contains(Permutation(3))
False
>>> H.contains(Permutation(3), strict=False)
True
```

To test if a given permutation is present in the group:

```
>>> elem in G.generators
False
>>> G.has(elem)
False
```

coset_factor(g, factor_index=False)

Return G's (self's) coset factorization of g

If g is an element of G then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in f are those for which the product gives g: $g = f[n] * \dots * f[1] * f[0]$ where $n = \text{len}(B)$ and $B = G.\text{base}$. $f[i]$ is one of the permutations in $\text{self}.\text{basic\_orbits}[i]$.

If factor_index==True, returns a tuple $[b[0], \dots, b[n]]$, where $b[i]$ belongs to $\text{self}.\text{basic\_orbits}[i]$

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> Permutation.print_cyclic = True
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
```

Define g:

```
>>> g = Permutation(7)(1, 2, 4)(3, 6, 5)
```

Confirm that it is an element of G:

```
>>> G.contains(g)
True
```

Thus, it can be written as a product of factors (up to 3) drawn from u. See below that a factor from u1 and u2 and the Identity permutation have been used:

```
>>> f = G.coset_factor(g)
>>> f[2]*f[1]*f[0] == g
True
>>> f1 = G.coset_factor(g, True); f1
[0, 4, 4]
>>> tr = G.basic_transversals
>>> f[0] == tr[0][f1[0]]
True
```

If g is not an element of G then [] is returned:

```
>>> c = Permutation(5, 6, 7)
>>> G.coset_factor(c)
[]
```

see `util._strip`

coset_rank(g)

rank using Schreier-Sims representation

The coset rank of g is the ordering number in which it appears in the lexicographic listing according to the coset decomposition

The ordering is the same as in `G.generate(method='coset')`. If g does not belong to the group it returns None.

See also:

[coset_factor](#) (page 239)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
(7)(2 4)(3 5)
```

coset_table(H)

Return the standardised (right) coset table of self in H as a list of lists.

coset_transversal(H)

Return a transversal of the right cosets of self by its subgroup H using the second method described in [1], Subsection 4.6.7

coset_unrank(rank, af=False)

unrank using Schreier-Sims representation

coset_unrank is the inverse operation of coset_rank if $0 \leq \text{rank} < \text{order}$; otherwise it returns None.

degree

Returns the size of the permutations in the group.

The number of permutations comprising the group is given by `len(group)`; the number of permutations that can be generated by the group is given by `group.order()`.

See also:

[order](#) (page 253)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (2)(0 1)]
```

derived_series()

Return the derived series for the group.

The derived series for a group G is defined as $G = G_0 > G_1 > G_2 > \dots$ where $G_i = [G_{i-1}, G_{i-1}]$, i.e. G_i is the derived subgroup of G_{i-1} , for $i \in \mathbb{N}$. When we have $G_k = G_{k-1}$ for some $k \in \mathbb{N}$, the series terminates.

Returns A list of permutation groups containing the members of the derived series in the order $G = G_0, G_1, G_2, \dots$

See also:

[derived_subgroup](#) (page 242)

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
```

(continues on next page)

(continued from previous page)

```

>>> S = SymmetricGroup(4)
>>> len(S.derived_series())
4
>>> S.derived_series()[1].is_subgroup(AlternatingGroup(4))
True
>>> S.derived_series()[2].is_subgroup(DihedralGroup(2))
True

```

derived_subgroup()

Compute the derived subgroup.

The derived subgroup, or commutator subgroup is the subgroup generated by all commutators $[g, h] = hgh^{-1}g^{-1}$ for $g, h \in G$; it is equal to the normal closure of the set of commutators of the generators ([1], p.28, [11]).

See also:

[derived_series](#) (page 241)

Examples

```

>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2, 4, 3])
>>> b = Permutation([0, 1, 3, 2, 4])
>>> G = PermutationGroup([a, b])
>>> C = G.derived_subgroup()
>>> list(C.generate(af=True))
[[0, 1, 2, 3, 4], [0, 1, 3, 4, 2], [0, 1, 4, 2, 3]]

```

elements

Returns all the elements of the permutation group as a set

Examples

```

>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p.elements
{(3), (2 3), (3)(1 2), (1 2 3), (1 3 2), (1 3)}

```

generate(method='coset', af=False)

Return iterator to generate the elements of the group

Iteration is done with one of these methods:

```

method='coset' using the Schreier-Sims coset representation
method='dimino' using the Dimino method

```

If `af = True` it yields the array form of the permutations

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics import PermutationGroup
>>> from sympy.combinatorics.polyhedron import tetrahedron
```

The permutation group given in the tetrahedron object is also true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group
True
```

Also the group generated by the permutations in the tetrahedron pgroup - even the first two - is a proper group:

```
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate())); J
PermutationGroup([
  (0 1)(2 3),
  (1 2 3),
  (1 3 2),
  (0 3 1),
  (0 2 3),
  (0 3)(1 2),
  (0 1 3),
  (3)(0 2 1),
  (0 3 2),
  (3)(0 1 2),
  (0 2)(1 3)])
>>> J.is_group
True
```

generate_dimino(af=False)

Yield group elements using Dimino's algorithm

If af == True it yields the array form of the permutations

References

[1] The Implementation of Various Algorithms for Permutation Groups in the Computer Algebra System: AXIOM, N.J. Doye, M.Sc. Thesis

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_dimino(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]
```

generate_schreier_sims(af=False)

Yield group elements using the Schreier-Sims representation in coset_rank order

If af = True it yields the array form of the permutations

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 3, 2, 1],
 [0, 1, 3, 2], [0, 2, 3, 1], [0, 3, 1, 2]]
```

generator_product(g, original=False)

Return a list of strong generators $[s_1, \dots, s_n]$ s.t $g = s_n * \dots * s_1$. If *original* = True, make the list contain only the original group generators

generators

Returns the generators of the group.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.generators
[(1 2), (2)(0 1)]
```

identity

Return the identity element of the permutation group.

is_abelian

Test if the group is Abelian.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.is_abelian
False
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
```

(continues on next page)

(continued from previous page)

```
>>> G.is_abelian
True
```

is_alt_sym(eps=0.05, _random_prec=None)

Monte Carlo test for the symmetric/alternating group for degrees ≥ 8 .

More specifically, it is one-sided Monte Carlo with the answer True (i.e., G is symmetric/alternating) guaranteed to be correct, and the answer False being incorrect with probability eps.

For degree < 8 , the order of the group is checked so the test is deterministic.

See also:

[_check_cycles_alt_sym](#)

Notes

The algorithm itself uses some nontrivial results from group theory and number theory: 1) If a transitive group G of degree n contains an element with a cycle of length $n/2 < p < n-2$ for p a prime, G is the symmetric or alternating group ([1], pp. 81-82) 2) The proportion of elements in the symmetric/alternating group having the property described in 1) is approximately $\log(2)/\log(n)$ ([1], p.82; [2], pp. 226-227). The helper function [_check_cycles_alt_sym](#) is used to go over the cycles in a permutation and look for ones satisfying 1).

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

is_nilpotent

Test if the group is nilpotent.

A group G is nilpotent if it has a central series of finite length. Alternatively, G is nilpotent if its lower central series terminates with the trivial group. Every nilpotent group is also solvable ([1], p.29, [12]).

See also:

[lower_central_series](#) (page 248), [is_solvable](#) (page 246)

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
```

(continues on next page)

(continued from previous page)

```
>>> S.is_nilpotent
False
```

is_normal(gr, strict=True)

Test if $G=\text{self}$ is a normal subgroup of gr.

G is normal in gr if for each g_2 in G , g_1 in gr, $g = g_1 g_2 g_1^{-1}$ belongs to G . It is sufficient to check this for each g_1 in gr.generators and g_2 in G .generators.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G1 = PermutationGroup([a, Permutation([2, 0, 1])])
>>> G1.is_normal(G)
True
```

is_primitive(randomized=True)

Test if a group is primitive.

A permutation group G acting on a set S is called primitive if S contains no nontrivial block under the action of G (a block is nontrivial if its cardinality is more than 1).

See also:

[minimal_block](#) (page 250), [random_stab](#) (page 254)

Notes

The algorithm is described in [1], p.83, and uses the function `minimal_block` to search for blocks of the form $\{0, k\}$ for k ranging over representatives for the orbits of G_0 , the stabilizer of 0. This algorithm has complexity $O(n^2)$ where n is the degree of the group, and will perform badly if G_0 is small.

There are two implementations offered: one finds G_0 deterministically using the function `stabilizer`, and the other (default) produces random elements of G_0 using `random_stab`, hoping that they generate a subgroup of G_0 with not too many more orbits than G_0 (this is suggested in [1], p.83). Behavior is changed by the `randomized` flag.

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False
```

is_solvable

Test if the group is solvable.

G is solvable if its derived series terminates with the trivial group ([1], p.29).

See also:

[is_nilpotent](#) (page 245), [derived_series](#) (page 241)

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True
```

is_subgroup(G, strict=True)

Return True if all elements of self belong to G.

If strict is False then if self's degree is smaller than G's, the elements will be resized to have the same degree.

Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
...         CyclicGroup)
```

Testing is strict by default: the degree of each group must be the same:

```
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G1 = PermutationGroup([Permutation(0, 1, 2), Permutation(0, 1)])
>>> G2 = PermutationGroup([Permutation(0, 2), Permutation(0, 1, 2)])
>>> G3 = PermutationGroup([p, p**2])
>>> assert G1.order() == G2.order() == G3.order() == 6
>>> G1.is_subgroup(G2)
True
>>> G1.is_subgroup(G3)
False
>>> G3.is_subgroup(PermutationGroup(G3[1]))
False
>>> G3.is_subgroup(PermutationGroup(G3[0]))
True
```

To ignore the size, set strict to False:

```
>>> S3 = SymmetricGroup(3)
>>> S5 = SymmetricGroup(5)
>>> S3.is_subgroup(S5, strict=False)
True
>>> C7 = CyclicGroup(7)
>>> G = S5*C7
>>> S5.is_subgroup(G, False)
True
>>> C7.is_subgroup(G, 0)
False
```

is_transitive(strict=True)

Test if the group is transitive.

A group is transitive if it has a single orbit.

If strict is False the group is transitive if it has a single orbit of length different from 1.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G1 = PermutationGroup([a, b])
>>> G1.is_transitive()
False
>>> G1.is_transitive(strict=False)
True
>>> c = Permutation([2, 3, 0, 1])
>>> G2 = PermutationGroup([a, c])
>>> G2.is_transitive()
True
>>> d = Permutation([1, 0, 2, 3])
>>> e = Permutation([0, 1, 3, 2])
>>> G3 = PermutationGroup([d, e])
>>> G3.is_transitive() or G3.is_transitive(strict=False)
False
```

is_trivial

Test if the group is the trivial group.

This is true if the group contains only the identity permutation.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True
```

lower_central_series()

Return the lower central series for the group.

The lower central series for a group G is the series $G = G_0 > G_1 > G_2 > \dots$ where $G_k = [G, G_{k-1}]$, i.e. every term after the first is equal to the commutator of G and the previous term in G ([1], p.29).

Returns A list of permutation groups in the order $G = G_0, G_1, G_2, \dots$

See also:[commutator](#) (page 238), [derived_series](#) (page 241)

Examples

```
>>> from sympy.combinatorics.named_groups import (AlternatingGroup,
... DihedralGroup)
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

make_perm(n, seed=None)

Multiply n randomly selected permutations from pgroup together, starting with the identity permutation. If n is a list of integers, those integers will be used to select the permutations and they will be applied in L to R order: make_perm((A, B, C)) will give CBA(I) where I is the identity permutation.

seed is used to set the seed for the random selection of permutations from pgroup. If this is a list of integers, the corresponding permutations from pgroup will be selected in the order give. This is mainly used for testing purposes.

See also:

[random](#) (page 254)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a, b = [Permutation([1, 0, 3, 2]), Permutation([1, 3, 0, 2])]
>>> G = PermutationGroup([a, b])
>>> G.make_perm(1, [0])
(0 1)(2 3)
>>> G.make_perm(3, [0, 1, 0])
(0 2 3 1)
>>> G.make_perm([0, 1, 0])
(0 2 3 1)
```

max_div

Maximum proper divisor of the degree of a permutation group.

See also:

[minimal_block](#) (page 250), [_union_find_merge](#)

Notes

Obviously, this is the degree divided by its minimal proper divisor (larger than 1, if one exists). As it is guaranteed to be prime, the sieve from sympy.ntheory is used. This function is also used as an optimization tool for the functions [minimal_block](#) and [_union_find_merge](#).

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2
```

`minimal_block(points)`

For a transitive group, finds the block system generated by points.

If a group G acts on a set S , a nonempty subset B of S is called a block under the action of G if for all g in G we have $gB = B$ (g fixes B) or gB and B have no common points (g moves B entirely). ([1], p.23; [6]).

The distinct translates gB of a block B for g in G partition the set S and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides $|S|$ ([1], p.23). A G -congruence is an equivalence relation $\sim$ on the set S such that $a \sim b$ implies $g(a) \sim g(b)$ for all g in G . For a transitive group, the equivalence classes of a G -congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the G -congruence generated by the pairs $(p_0, p_1), (p_0, p_2), \dots, (p_0, p_{k-1})$ which is the same as finding the maximal block system (i.e., the one with minimum block size) such that $p_0, \dots, p_{k-1}$ are in the same block ([1], p.83).

It is an implementation of Atkinson's algorithm, as suggested in [1], and manipulates an equivalence relation on the set S using a union-find data structure. The running time is just above $O(|points||S|)$. ([1], pp. 83-87; [7]).

See also:

`_union_find_rep`, `_union_find_merge`, `is_transitive` (page 247), `is_primitive` (page 246)

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.minimal_block([0, 5])
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
>>> D.minimal_block([0, 1])
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

`minimal_blocks(randomized=True)`

For a transitive group, return the list of all minimal block systems. If a group is intransitive, return *False*.

See also:

`minimal_block` (page 250), `is_transitive` (page 247), `is_primitive` (page 246)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> DihedralGroup(6).minimal_blocks()
[[0, 1, 0, 1, 0, 1], [0, 1, 2, 0, 1, 2]]
>>> G = PermutationGroup(Permutation(1,2,5))
>>> G.minimal_blocks()
False
```

normal_closure(other, k=10)

Return the normal closure of a subgroup/set of permutations.

If S is a subset of a group G , the normal closure of A in G is defined as the intersection of all normal subgroups of G that contain A ([1], p.14). Alternatively, it is the group generated by the conjugates $x^{-1}yx$ for x a generator of G and y a generator of the subgroup $\langle S \rangle$ (for some chosen generating set for $\langle S \rangle$) ([1], p.73).

Parameters other

a subgroup/list of permutations/single permutation

k

an implementation-specific parameter that determines the number of conjugates that are adjoined to other at once

See also:

[commutator](#) (page 238), [derived_subgroup](#) (page 242), [random_pr](#) (page 254)

Notes

The algorithm is described in [1], pp. 73-74; it makes use of the generation of random elements for permutation groups by the product replacement algorithm.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup, AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```

orbit(alpha, action='tuples')

Compute the orbit of alpha $\{g(\alpha) | g \in G\}$ as a set.

The time complexity of the algorithm used here is $O(|Orb| * r)$ where $|Orb|$ is the size of the orbit and r is the number of generators of the group. For a more detailed analysis, see [1], p.78, [2], pp. 19-21. Here alpha can be a single point, or a list of points.

If alpha is a single point, the ordinary orbit is computed. if alpha is a list of points, there are three available options:

‘union’ - computes the union of the orbits of the points in the list ‘tuples’ - computes the orbit of the list interpreted as an ordered tuple under the group action (i.e., $g((1,2,3)) = (g(1), g(2), g(3))$) ‘sets’ - computes the orbit of the list interpreted as a sets

See also:

[orbit_transversal](#) (page 252)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0, 4, 5, 6, 3])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
{0, 1, 2}
>>> G.orbit([0, 4], 'union')
{0, 1, 2, 3, 4, 5, 6}
```

orbit_rep(alpha, beta, schreier_vector=None)

Return a group element which sends alpha to beta.

If beta is not in the orbit of alpha, the function returns False. This implementation makes use of the schreier vector. For a proof of correctness, see [1], p.80

See also:

[schreier_vector](#) (page 257)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
(0 4 1 2 3)
```

orbit_transversal(alpha, pairs=False)

Computes a transversal for the orbit of alpha as a set.

For a permutation group G , a transversal for the orbit $Orb = \{g(\alpha) | g \in G\}$ is a set $\{g_\beta | g_\beta(\alpha) = \beta\}$ for $\beta \in Orb$. Note that there may be more than one possible transversal. If pairs is set to True, it returns the list of pairs (β, g_β) . For a proof of correctness, see [1], p.79

See also:

[orbit](#) (page 251)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.orbit_transversal(0)
[(5), (0 1 2 3 4 5), (0 5)(1 4)(2 3), (0 2 4)(1 3 5), (5)(0 4)(1 3), (0 3)(1
↪4)(2 5)]
```

orbits(rep=False)

Return the orbits of self, ordered according to lowest element in each orbit.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[{0, 2, 3, 4, 6}, {1, 5}]
```

order()

Return the order of the group: the number of permutations that can be generated from elements of the group.

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

See also:

[degree](#) (page 241)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
```

```
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (2)(0 1)]
```

```
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
```

(continues on next page)

(continued from previous page)

```
>>> G = PermutationGroup([a, b])
>>> G.order()
6
```

pointwise_stabilizer(points, incremental=True)

Return the pointwise stabilizer for a set of points.

For a permutation group G and a set of points $\{p_1, p_2, \dots, p_k\}$, the pointwise stabilizer of $p_1, p_2, \dots, p_k$ is defined as $G_{p_1, \dots, p_k} = \{g \in G \mid g(p_i) = p_i \forall i \in \{1, 2, \dots, k\}\}$ ([1], p20). It is a subgroup of G .

See also:

[stabilizer](#) (page 257), [schreier_sims_incremental](#) (page 255)

Notes

When `incremental == True`, rather than the obvious implementation using successive calls to `.stabilizer()`, this uses the incremental Schreier-Sims algorithm to obtain a base with starting segment - the given points.

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True
```

presentation(eliminate_gens=True)

Return an *FpGroup* presentation of the group.

The algorithm is described in [1], Chapter 6.1.

random(af=False)

Return a random group element

random_pr(gen_count=11, iterations=50, _random_prec=None)

Return a random group element using product replacement.

For the details of the product replacement algorithm, see `_random_pr_init`. In `random_pr` the actual 'product replacement' is performed. Notice that if the attribute `_random_gens` is empty, it needs to be initialized by `_random_pr_init`.

See also:

`_random_pr_init`

random_stab(alpha, schreier_vector=None, _random_prec=None)

Random element from the stabilizer of `alpha`.

The schreier vector for `alpha` is an optional argument used for speeding up repeated calls. The algorithm is described in [1], p.81

See also:

[random_pr](#) (page 254), [orbit_rep](#) (page 252)

schreier_sims()

Schreier-Sims algorithm.

It computes the generators of the chain of stabilizers $G > G_{b_1} > \dots > G_{b_1, \dots, b_r} > 1$ in which $G_{b_1, \dots, b_i}$ stabilizes $b_1, \dots, b_i$, and the corresponding s cosets. An element of the group can be written as the product $h_1 * \dots * h_s$.

We use the incremental Schreier-Sims algorithm.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_sims()
>>> G.basic_transversals
[{0: (2)(0 1), 1: (2), 2: (1 2)},
 {0: (2), 2: (0 2)}]
```

schreier_sims_incremental(base=None, gens=None, slp_dict=False)

Extend a sequence of points and generating set to a base and strong generating set.

Parameters

base
The sequence of points to be extended to a base. Optional parameter with default value [].

gens

The generating set to be extended to a strong generating set relative to the base obtained. Optional parameter with default value `self.generators`.

slp_dict

If `True`, return a dictionary $g : gens$ for each strong generator g where $gens$ is a list of strong generators coming before g in `strong_gens`, such that the product of the elements of $gens$ is equal to g .

Returns (base, strong_gens)

`base` is the base obtained, and `strong_gens` is the strong generating set relative to it. The original parameters `base`, `gens` remain unchanged.

See also:

[schreier_sims](#) (page 254), [schreier_sims_random](#) (page 256)

Notes

This version of the Schreier-Sims algorithm runs in polynomial time. There are certain assumptions in the implementation - if the trivial group is provided, `base` and `gens` are returned immediately, as any sequence of points is a base for the trivial group. If the identity is present in the generators `gens`, it is removed as it is a redundant generator. The implementation is described in [1], pp. 90-93.

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(7)
>>> base = [2, 3]
>>> seq = [2, 3]
>>> base, strong_gens = A.schreier_sims_incremental(base=seq)
>>> _verify_bsgs(A, base, strong_gens)
True
>>> base[:2]
[2, 3]
```

schreier_sims_random(base=None, gens=None, consec_succ=10, _random_prec=None)
Randomized Schreier-Sims algorithm.

The randomized Schreier-Sims algorithm takes the sequence `base` and the generating set `gens`, and extends `base` to a base, and `gens` to a strong generating set relative to that base with probability of a wrong answer at most $2^{-\text{consec\_succ}}$, provided the random generators are sufficiently random.

Parameters

base
The sequence to be extended to a base.

gens

The generating set to be extended to a strong generating set.

consec_succ

The parameter defining the probability of a wrong answer.

_random_prec

An internal parameter used for testing purposes.

Returns

 (base, strong_gens)

`base` is the base and `strong_gens` is the strong generating set relative to it.

See also:

[schreier_sims](#) (page 254)

Notes

The algorithm is described in detail in [1], pp. 97-98. It extends the orbits `orbs` and the permutation groups `stabs` to basic orbits and basic stabilizers for the base and strong generating set produced in the end. The idea of the extension process is to “sift” random group elements through the stabilizer chain and amend the stabilizers/orbits along the way when a sift is not successful. The helper function `_strip` is used to attempt to decompose a random group element according to the current state of the stabilizer chain and report whether the element was fully decomposed (successful sift) or not (unsuccessful sift). In the latter case, the level at which the sift failed is reported and used to amend `stabs`, `base`, `gens` and `orbs` accordingly. The halting condition is for `consec_succ` consecutive successful sifts to pass. This

makes sure that the current base and gens form a BSGS with probability at least $1 - 1/\text{consec\_succ}$.

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

schreier_vector(alpha)

Computes the schreier vector for alpha.

The Schreier vector efficiently stores information about the orbit of alpha. It can later be used to quickly obtain elements of the group that send alpha to a particular element in the orbit. Notice that the Schreier vector depends on the order in which the group generators are listed. For a definition, see [3]. Since list indices start from zero, we adopt the convention to use "None" instead of 0 to signify that an element doesn't belong to the orbit. For the algorithm and its correctness, see [2], pp.78-80.

See also:

[orbit](#) (page 251)

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([2, 4, 6, 3, 1, 5, 0])
>>> b = Permutation([0, 1, 3, 5, 4, 6, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_vector(0)
[-1, None, 0, 1, None, 1, 0]
```

stabilizer(alpha)

Return the stabilizer subgroup of alpha.

The stabilizer of α is the group $G_\alpha = \{g \in G | g(\alpha) = \alpha\}$. For a proof of correctness, see [1], p.79.

See also:

[orbit](#) (page 251)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
```

(continues on next page)

(continued from previous page)

```
>>> G.stabilizer(5)
PermutationGroup([
    (5)(0 4)(1 3)])
```

strong_gens

Return a strong generating set from the Schreier-Sims algorithm.

A generating set $S = \{g_1, g_2, \dots, g_t\}$ for a permutation group G is a strong generating set relative to the sequence of points (referred to as a “base”) $(b_1, b_2, \dots, b_k)$ if, for $1 \leq i \leq k$ we have that the intersection of the pointwise stabilizer $G^{(i+1)} := G_{b_1, b_2, \dots, b_i}$ with S generates the pointwise stabilizer $G^{(i+1)}$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

See also:

[base](#) (page 234), [basic_transversals](#) (page 236), [basic_orbits](#) (page 235), [basic_stabilizers](#) (page 236)

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> D.strong_gens
[(0 1 2 3), (0 3)(1 2), (1 3)]
>>> D.base
[0, 1]
```

strong_presentation()

Return a strong finite presentation of G . The generators of the returned group are in the same order as the strong generators of G .

The algorithm is based on Sims’ Verify algorithm described in [1], Chapter 6.

See also:

[presentation](#) (page 254), [_verify](#)

Example

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> P = DihedralGroup(4)
>>> G = P.strong_presentation()
>>> P.order() == G.order()
True
```

subgroup(gens)

Return the subgroup generated by *gens* which is a list of elements of the group

subgroup_search(prop, base=None, strong_gens=None, tests=None, init_subgroup=None)

Find the subgroup of all elements satisfying the property prop.

This is done by a depth-first search with respect to base images that uses several tests to prune the search tree.

Parameters `prop`

The property to be used. Has to be callable on group elements and always return True or False. It is assumed that all group elements satisfying `prop` indeed form a subgroup.

base

A base for the supergroup.

strong_gens

A strong generating set for the supergroup.

tests

A list of callables of length equal to the length of `base`. These are used to rule out group elements by partial base images, so that `tests[l](g)` returns False if the element `g` is known not to satisfy `prop` base on where `g` sends the first `l + 1` base points.

init_subgroup

if a subgroup of the sought group is known in advance, it can be passed to the function as this parameter.

Returns `res`

The subgroup of all elements satisfying `prop`. The generating set for this group is guaranteed to be a strong generating set relative to the base `base`.

Notes

This function is extremely lengthy and complicated and will require some careful attention. The implementation is described in [1], pp. 114-117, and the comments for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits all members of the supergroup. However, there are a lot of tests which are used to prune the search tree, and users can define their own tests via the `tests` parameter, so in practice, and for some computations, it's not terrible.

A crucial part in the procedure is the frequent base change performed (this is line 11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentions that this can be done by using `.baseswap(...)`, however the current implementation uses a more straightforward way to find the next basic stabilizer - calling the function `.stabilizer(...)` on the previous basic stabilizer.

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(7)
>>> prop_even = lambda x: x.is_even
>>> base, strong_gens = S.schreier_sims_incremental()
>>> G = S.subgroup_search(prop_even, base=base, strong_gens=strong_gens)
```

(continues on next page)

(continued from previous page)

```
>>> G.is_subgroup(AlternatingGroup(7))
True
>>> _verify_bsgs(G, base, G.generators)
True
```

`syLOW_subgroup(p)`

Return a p-SyLOW subgroup of the group.

The algorithm is described in [1], Chapter 4, Section 7

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import AlternatingGroup
```

```
>>> D = DihedralGroup(6)
>>> S = D.syLOW_subgroup(2)
>>> S.order()
4
>>> G = SymmetricGroup(6)
>>> S = G.syLOW_subgroup(5)
>>> S.order()
5
```

```
>>> G1 = AlternatingGroup(3)
>>> G2 = AlternatingGroup(5)
>>> G3 = AlternatingGroup(9)
```

```
>>> S1 = G1.syLOW_subgroup(3)
>>> S2 = G2.syLOW_subgroup(3)
>>> S3 = G3.syLOW_subgroup(3)
```

```
>>> len1 = len(S1.lower_central_series())
>>> len2 = len(S2.lower_central_series())
>>> len3 = len(S3.lower_central_series())
```

```
>>> len1 == len2
True
>>> len1 < len3
True
```

`transitivity_degree`

Compute the degree of transitivity of the group.

A permutation group G acting on $\Omega = \{0, 1, \dots, n - 1\}$ is k -fold transitive, if, for any k points $(a_1, a_2, \dots, a_k) \in \Omega$ and any k points $(b_1, b_2, \dots, b_k) \in \Omega$ there exists $g \in G$ such that $g(a_1) = b_1, g(a_2) = b_2, \dots, g(a_k) = b_k$. The degree of transitivity of G is the maximum k such that G is k -fold transitive. ([8])

See also:

[is_transitive](#) (page 247), [orbit](#) (page 251)

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```

Polyhedron

class sympy.combinatorics.polyhedron.**Polyhedron**

Represents the polyhedral symmetry group (PSG).

The PSG is one of the symmetry groups of the Platonic solids. There are three polyhedral groups: the tetrahedral group of order 12, the octahedral group of order 24, and the icosahedral group of order 60.

All doctests have been given in the docstring of the constructor of the object.

References

<http://mathworld.wolfram.com/PolyhedralGroup.html>

array_form

Return the indices of the corners.

The indices are given relative to the original position of corners.

See also:

[corners](#) (page 261), [cyclic_form](#) (page 262)

Examples

```
>>> from sympy.combinatorics import Permutation, Cycle
>>> from sympy.combinatorics.polyhedron import tetrahedron
>>> tetrahedron.array_form
[0, 1, 2, 3]
```

```
>>> tetrahedron.rotate(0)
>>> tetrahedron.array_form
[0, 2, 3, 1]
>>> tetrahedron.pgroup[0].array_form
[0, 2, 3, 1]
```

corners

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

See also:

[array_form](#) (page 261), [cyclic_form](#) (page 262)

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

cyclic_form

Return the indices of the corners in cyclic notation.

The indices are given relative to the original position of corners.

See also:

[corners](#) (page 261), [array_form](#) (page 261)

edges

Given the faces of the polyhedra we can get the edges.

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
{(0, 1), (0, 2), (1, 2)}
```

faces

Get the faces of the Polyhedron.

pgroup

Get the permutations of the Polyhedron.

reset()

Return corners to their original positions.

Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron as T
>>> T.corners
(0, 1, 2, 3)
>>> T.rotate(0)
>>> T.corners
(0, 2, 3, 1)
>>> T.reset()
>>> T.corners
(0, 1, 2, 3)
```

rotate(perm)

Apply a permutation to the polyhedron in place. The permutation may be given as a `Permutation` instance or an integer indicating which permutation from `pgroup` of the `Polyhedron` should be applied.

This is an operation that is analogous to rotation about an axis by a fixed increment.

Notes

When a `Permutation` is applied, no check is done to see if that is a valid permutation for the `Polyhedron`. For example, a cube could be given a permutation which effectively swaps only 2 vertices. A valid permutation (that rotates the object in a physical way) will be obtained if one only uses permutations from the `pgroup` of the `Polyhedron`. On the other hand, allowing arbitrary rotations (applications of permutations) gives a way to follow named elements rather than indices since `Polyhedron` allows vertices to be named while `Permutation` works only with indices.

Examples

```
>>> from sympy.combinatorics import Polyhedron, Permutation
>>> from sympy.combinatorics.polyhedron import cube
>>> cube.corners
(0, 1, 2, 3, 4, 5, 6, 7)
>>> cube.rotate(0)
>>> cube.corners
(1, 2, 3, 0, 5, 6, 7, 4)
```

A non-physical “rotation” that is not prohibited by this method:

```
>>> cube.reset()
>>> cube.rotate(Permutation([[1, 2]], size=8))
>>> cube.corners
(0, 2, 1, 3, 4, 5, 6, 7)
```

`Polyhedron` can be used to follow elements of set that are identified by letters instead of integers:

```
>>> shadow = h5 = Polyhedron(list('abcde'))
>>> p = Permutation([3, 0, 1, 2, 4])
>>> h5.rotate(p)
>>> h5.corners
(d, a, b, c, e)
>>> _ == shadow.corners
True
>>> copy = h5.copy()
>>> h5.rotate(p)
>>> h5.corners == copy.corners
False
```

size

Get the number of corners of the `Polyhedron`.

vertices

Get the corners of the `Polyhedron`.

The method `vertices` is an alias for `corners`.

See also:

[array_form](#) (page 261), [cyclic_form](#) (page 262)

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

Prufer Sequences

`class sympy.combinatorics.prufer.Prufer`

The Prufer correspondence is an algorithm that describes the bijection between labeled trees and the Prufer code. A Prufer code of a labeled tree is unique up to isomorphism and has a length of $n - 2$.

Prufer sequences were first used by Heinz Prufer to give a proof of Cayley's formula.

References

[R36] (page 1855)

`static edges(*runs)`

Return a list of edges and the number of nodes from the given runs that connect nodes in an integer-labelled tree.

All node numbers will be shifted so that the minimum node is 0. It is not a problem if edges are repeated in the runs; only unique edges are returned. There is no assumption made about what the range of the node labels should be, but all nodes from the smallest through the largest must be present.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.edges([1, 2, 3], [2, 4, 5]) # a T
([[0, 1], [1, 2], [1, 3], [3, 4]], 5)
```

Duplicate edges are removed:

```
>>> Prufer.edges([0, 1, 2, 3], [1, 4, 5], [1, 4, 6]) # a K
([[0, 1], [1, 2], [1, 4], [2, 3], [4, 5], [4, 6]], 7)
```

`next(delta=1)`

Generates the Prufer sequence that is delta beyond the current one.

See also:

[prufer_rank](#) (page 265), [rank](#) (page 266), [prev](#) (page 265), [size](#) (page 266)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> b = a.next(1) # == a.next()
>>> b.tree_repr
[[0, 2], [0, 1], [1, 3]]
>>> b.rank
1
```

nodes

Returns the number of nodes in the tree.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).nodes
6
>>> Prufer([1, 0, 0]).nodes
5
```

prev(delta=1)

Generates the Prufer sequence that is -delta before the current one.

See also:

[prufer_rank](#) (page 265), [rank](#) (page 266), [next](#) (page 264), [size](#) (page 266)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer([1, 2, 0])
>>> b.rank
35
```

prufer_rank()

Computes the rank of a Prufer sequence.

See also:

[rank](#) (page 266), [next](#) (page 264), [prev](#) (page 265), [size](#) (page 266)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

prufer_repr

Returns Prufer sequence for the Prufer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prufer sequence is the list of recorded nodes.

See also:

[to_prufer](#) (page 266)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).prufer_repr
[3, 3, 3, 4]
>>> Prufer([1, 0, 0]).prufer_repr
[1, 0, 0]
```

rank

Returns the rank of the Prufer sequence.

See also:

[prufer_rank](#) (page 265), [next](#) (page 264), [prev](#) (page 265), [size](#) (page 266)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

size

Return the number of possible trees of this Prufer object.

See also:

[prufer_rank](#) (page 265), [rank](#) (page 266), [next](#) (page 264), [prev](#) (page 265)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([0]*4).size == Prufer([6]*4).size == 1296
True
```

static to_prufer(tree, n)

Return the Prufer sequence for a tree given as a list of edges where n is the number of nodes in the tree.

See also:

prufer_repr (page 265) returns Prufer sequence of a Prufer object.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_repr
[0, 0]
>>> Prufer.to_prufer([[0, 1], [0, 2], [0, 3]], 4)
[0, 0]
```

static to_tree(prufer)

Return the tree (as a list of edges) of the given Prufer sequence.

See also:

tree_repr (page 267) returns tree representation of a Prufer object.

References

- <https://hamberg.no/erlend/posts/2010-11-06-prufer-sequence-compact-tree-representation.html>

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([0, 2], 4)
>>> a.tree_repr
[[0, 1], [0, 2], [2, 3]]
>>> Prufer.to_tree([0, 2])
[[0, 1], [0, 2], [2, 3]]
```

tree_repr

Returns the tree representation of the Prufer object.

See also:

to_tree (page 267)

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).tree_repr
[[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]
>>> Prufer([1, 0, 0]).tree_repr
[[1, 2], [0, 1], [0, 3], [0, 4]]
```

classmethod unrank(rank, n)

Finds the unranked Prufer sequence.

Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.unrank(0, 4)
Prufer([0, 0])
```

Subsets

class `sympy.combinatorics.subsets.Subset`

Represents a basic subset object.

We generate subsets using essentially two techniques, binary enumeration and lexicographic enumeration. The Subset class takes two arguments, the first one describes the initial subset to consider and the second describes the superset.

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a.prev_binary().subset
['c']
```

classmethod `bitlist_from_subset(subset, superset)`

Gets the bitlist corresponding to a subset.

See also:

[subset_from_bitlist](#) (page 272)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.bitlist_from_subset(['c', 'd'], ['a', 'b', 'c', 'd'])
'0011'
```

cardinality

Returns the number of all possible subsets.

See also:

[subset](#) (page 272), [superset](#) (page 273), [size](#) (page 272), [superset_size](#) (page 273)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.cardinality
16
```

iterate_binary(k)

This is a helper function. It iterates over the binary subsets by k steps. This variable can be both positive or negative.

See also:

[next_binary](#) (page 269), [prev_binary](#) (page 270)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(-2).subset
['d']
>>> a = Subset(['a', 'b', 'c'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(2).subset
[]
```

iterate_graycode(k)

Helper function used for [prev_gray](#) and [next_gray](#). It performs k step overs to get the respective Gray codes.

See also:

[next_gray](#) (page 269), [prev_gray](#) (page 270)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

next_binary()

Generates the next binary ordered subset.

See also:

[prev_binary](#) (page 270), [iterate_binary](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a = Subset(['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
[]
```

next_gray()

Generates the next Gray code ordered subset.

See also:

[iterate_graycode](#) (page 269), [prev_gray](#) (page 270)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.next_gray().subset
[1, 3]
```

`next_lexicographic()`

Generates the next lexicographically ordered subset.

See also:

[prev_lexicographic](#) (page 271)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
[]
```

`prev_binary()`

Generates the previous binary ordered subset.

See also:

[next_binary](#) (page 269), [iterate_binary](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['a', 'b', 'c', 'd']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['c']
```

`prev_gray()`

Generates the previous Gray code ordered subset.

See also:

[iterate_graycode](#) (page 269), [next_gray](#) (page 269)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([2, 3, 4], [1, 2, 3, 4, 5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

prev_lexicographic()

Generates the previous lexicographically ordered subset.

See also:

[next_lexicographic](#) (page 270)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['d']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['c']
```

rank_binary

Computes the binary ordered rank.

See also:

[iterate_binary](#) (page 268), [unrank_binary](#) (page 273)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

rank_gray

Computes the Gray code ranking of the subset.

See also:

[iterate_graycode](#) (page 269), [unrank_gray](#) (page 274)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_gray
2
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_gray
27
```

rank_lexicographic

Computes the lexicographic ranking of the subset.

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_lexicographic
14
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_lexicographic
43
```

size

Gets the size of the subset.

See also:

[subset](#) (page 272), [superset](#) (page 273), [superset_size](#) (page 273), [cardinality](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.size
2
```

subset

Gets the subset represented by the current instance.

See also:

[superset](#) (page 273), [size](#) (page 272), [superset_size](#) (page 273), [cardinality](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.subset
['c', 'd']
```

classmethod subset_from_bitlist(super_set, bitlist)

Gets the subset defined by the bitlist.

See also:

[bitlist_from_subset](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.subset_from_bitlist(['a', 'b', 'c', 'd'], '0011').subset
['c', 'd']
```

classmethod `subset_indices(subset, superset)`

Return indices of subset in superset in a list; the list is empty if all elements of subset are not in superset.

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> superset = [1, 3, 2, 5, 4]
>>> Subset.subset_indices([3, 2, 1], superset)
[1, 2, 0]
>>> Subset.subset_indices([1, 6], superset)
[]
>>> Subset.subset_indices([], superset)
[]
```

superset

Gets the superset of the subset.

See also:

[subset](#) (page 272), [size](#) (page 272), [superset_size](#) (page 273), [cardinality](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset
['a', 'b', 'c', 'd']
```

superset_size

Returns the size of the superset.

See also:

[subset](#) (page 272), [superset](#) (page 273), [size](#) (page 272), [cardinality](#) (page 268)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset_size
4
```

classmethod `unrank_binary(rank, superset)`

Gets the binary ordered subset of the specified rank.

See also:

[iterate_binary](#) (page 268), [rank_binary](#) (page 271)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.unrank_binary(4, ['a', 'b', 'c', 'd']).subset
['b']
```

classmethod `unrank_gray`(rank, superset)

Gets the Gray code ordered subset of the specified rank.

See also:

[iterate_graycode](#) (page 269), [rank_gray](#) (page 271)

Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.unrank_gray(4, ['a', 'b', 'c']).subset
['a', 'b']
>>> Subset.unrank_gray(0, ['a', 'b', 'c']).subset
[]
```

`subsets.ksubsets(k)`

Finds the subsets of size k in lexicographic order.

This uses the `itertools` generator.

See also:

class `Subset`

Examples

```
>>> from sympy.combinatorics.subsets import ksubsets
>>> list(ksubsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(ksubsets([1, 2, 3, 4, 5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

Gray Code

class `sympy.combinatorics.graycode.GrayCode`

A Gray code is essentially a Hamiltonian walk on a n-dimensional cube with edge length of one. The vertices of the cube are represented by vectors whose values are binary. The Hamilton walk visits each vertex exactly once. The Gray code for a 3d cube is ['000', '100', '110', '010', '011', '111', '101', '001'].

A Gray code solves the problem of sequentially generating all possible subsets of n objects in such a way that each subset is obtained from the previous one by either deleting or adding a single object. In the above example, 1 indicates that the object is present, and 0 indicates that its absent.

Gray codes have applications in statistics as well when we want to compute various statistics related to subsets in an efficient manner.

References: [1] Nijenhuis, A. and Wilf, H.S. (1978). Combinatorial Algorithms. Academic Press. [2] Knuth, D. (2011). The Art of Computer Programming, Vol 4 Addison Wesley

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100', '1100', '1101',
 '1111', '1110', '1010', '1011', '1001', '1000']
```

current

Returns the currently referenced Gray code as a bit string.

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(3, start='100').current
'100'
```

generate_gray(**hints)

Generates the sequence of bit vectors of a Gray Code.

[1] Knuth, D. (2011). The Art of Computer Programming, Vol 4, Addison Wesley

See also:

[skip](#) (page 276)

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(start='011'))
['011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(rank=4))
['110', '111', '101', '100']
```

n

Returns the dimension of the Gray code.

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(5)
>>> a.n
5
```

next(delta=1)

Returns the Gray code a distance delta (default = 1) from the current value in canonical order.

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

rank

Ranks the Gray code.

A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects w.r.t. a given order. For example, the 4 bit binary reflected Gray code (BRGC) '0101' has a rank of 6 as it appears in the 6th position in the canonical ordering of the family of 4 bit Gray codes.

References: [1] <http://statweb.stanford.edu/~susan/courses/s208/node12.html>

See also:

[unrank](#) (page 277)

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> GrayCode(3, start='100').rank
7
>>> GrayCode(3, rank=7).current
'100'
```

selections

Returns the number of bit vectors in the Gray code.

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> a.selections
8
```

skip()

Skips the bit generation.

See also:

[generate_gray](#) (page 275)

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...     if i == '010':
...         a.skip()
...     print(i)
...
000
001
011
010
111
101
100
```

classmethod unrank(n, rank)

Unranks an n-bit sized Gray code of rank k. This method exists so that a derivative GrayCode class can define its own code of a given rank.

The string here is generated in reverse order to allow for tail-call optimization.

See also:

[rank](#) (page 276)

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

graycode.random_bitstring()

Generates a random bitlist of length n.

Examples

```
>>> from sympy.combinatorics.graycode import random_bitstring
>>> random_bitstring(3)
100
```

graycode.gray_to_bin()

Convert from Gray coding to binary coding.

We assume big endian encoding.

See also:[bin_to_gray](#) (page 278)**Examples**

```
>>> from sympy.combinatorics.graycode import gray_to_bin
>>> gray_to_bin('100')
'111'
```

graycode.bin_to_gray()

Convert from binary coding to gray coding.

We assume big endian encoding.

See also:[gray_to_bin](#) (page 277)**Examples**

```
>>> from sympy.combinatorics.graycode import bin_to_gray
>>> bin_to_gray('111')
'100'
```

graycode.get_subset_from_bitstring(bitstring)

Gets the subset defined by the bitstring.

See also:[graycode_subsets](#) (page 278)**Examples**

```
>>> from sympy.combinatorics.graycode import get_subset_from_bitstring
>>> get_subset_from_bitstring(['a', 'b', 'c', 'd'], '0011')
['c', 'd']
>>> get_subset_from_bitstring(['c', 'a', 'c', 'c'], '1100')
['c', 'a']
```

graycode.graycode_subsets()

Generates the subsets as enumerated by a Gray code.

See also:[get_subset_from_bitstring](#) (page 278)**Examples**

```
>>> from sympy.combinatorics.graycode import graycode_subsets
>>> list(graycode_subsets(['a', 'b', 'c']))
[[], ['c'], ['b', 'c'], ['b'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'c'], ['a']]
>>> list(graycode_subsets(['a', 'b', 'c', 'c']))
[[[], ['c'], ['c', 'c'], ['c'], ['b', 'c'], ['b', 'c', 'c'], ['b', 'c'], ['b'],
→ ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'c'], ['a', (continues on next page)
→ 'c'], ['a', 'c', 'c'], ['a', 'c'], ['a']]
```

(continued from previous page)

Named Groups

`sympy.combinatorics.named_groups.SymmetricGroup(n)`

Generates the symmetric group on n elements as a permutation group.

The generators taken are the n -cycle $(0\ 1\ 2\ \dots\ n-1)$ and the transposition $(0\ 1)$ (in cycle notation). (See [1]). After the group is generated, some of its basic properties are set.

See also:

[CyclicGroup](#) (page 279), [DihedralGroup](#) (page 280), [AlternatingGroup](#) (page 280)

References

[1] http://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> G.is_group
True
>>> G.order()
24
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],
 [1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],
 [2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],
 [3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
 [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1]]
```

`sympy.combinatorics.named_groups.CyclicGroup(n)`

Generates the cyclic group of order n as a permutation group.

The generator taken is the n -cycle $(0\ 1\ 2\ \dots\ n-1)$ (in cycle notation). After the group is generated, some of its basic properties are set.

See also:

[SymmetricGroup](#) (page 279), [DihedralGroup](#) (page 280), [AlternatingGroup](#) (page 280)

Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(6)
>>> G.is_group
True
>>> G.order()
6
```

(continues on next page)

(continued from previous page)

```
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 0], [2, 3, 4, 5, 0, 1],
 [3, 4, 5, 0, 1, 2], [4, 5, 0, 1, 2, 3], [5, 0, 1, 2, 3, 4]]
```

sympy.combinatorics.named_groups.DihedralGroup(n)Generates the dihedral group D_n as a permutation group.

The dihedral group D_n is the group of symmetries of the regular n -gon. The generators taken are the n -cycle $a = (0\ 1\ 2\ \dots\ n-1)$ (a rotation of the n -gon) and $b = (0\ n-1)(1\ n-2)\dots$ (a reflection of the n -gon) in cycle rotation. It is easy to see that these satisfy $a^n = b^2 = 1$ and $bab = a^{-1}$ so they indeed generate D_n (See [1]). After the group is generated, some of its basic properties are set.

See also:[SymmetricGroup](#) (page 279), [CyclicGroup](#) (page 279), [AlternatingGroup](#) (page 280)**References**

[1] http://en.wikipedia.org/wiki/Dihedral_group

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> [perm.cyclic_form for perm in a]
[[[]], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]],
 [[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]],
 [[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]],
 [[0, 3], [1, 2]]]
```

sympy.combinatorics.named_groups.AlternatingGroup(n)Generates the alternating group on n elements as a permutation group.

For $n > 2$, the generators taken are $(0\ 1\ 2)$, $(0\ 1\ 2\ \dots\ n-1)$ for n odd and $(0\ 1\ 2)$, $(1\ 2\ \dots\ n-1)$ for n even (See [1], p.31, ex.6.9.). After the group is generated, some of its basic properties are set. The cases $n = 1, 2$ are handled separately.

See also:[SymmetricGroup](#) (page 279), [CyclicGroup](#) (page 279), [DihedralGroup](#) (page 280)**References**

[1] Armstrong, M. "Groups and Symmetry"

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> len(a)
12
>>> all(perm.is_even for perm in a)
True
```

`sympy.combinatorics.named_groups.AbelianGroup(*cyclic_orders)`

Returns the direct product of cyclic groups with the given orders.

According to the structure theorem for finite abelian groups ([1]), every finite abelian group can be written as the direct product of finitely many cyclic groups.

See also:

`DirectProduct`

References

[1] http://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> AbelianGroup(3, 4)
PermutationGroup([
    (6)(0 1 2),
    (3 4 5 6)])
>>> _.is_group
True
```

Utilities

`sympy.combinatorics.util._base_ordering(base, degree)`

Order $\{0, 1, \dots, n-1\}$ so that base points come first and in order.

Parameters “base” - the base

“degree” - the degree of the associated permutation group

Returns A list `base_ordering` such that `base_ordering[point]` is the number of point in the ordering.

Notes

This is used in backtrack searches, when we define a relation $<<$ on the underlying set for a permutation group of degree n , $\{0, 1, \dots, n-1\}$, so that if $(b_1, b_2, \dots, b_k)$ is a base we have $b_i << b_j$ whenever $i < j$ and $b_i << a$ for all $i \in \{1, 2, \dots, k\}$ and a is not in the base. The idea is developed and applied to backtracking algorithms in [1], pp.108-132. The points that are not in the base are taken in increasing order.

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _base_ordering
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> _base_ordering(S.base, S.degree)
[0, 1, 2, 3]
```

`sympy.combinatorics.util._check_cycles_alt_sym(perm)`

Checks for cycles of prime length p with $n/2 < p < n-2$.

Here n is the degree of the permutation. This is a helper function for the function `is_alt_sym` from `sympy.combinatorics.perm_groups`.

See also:

`sympy.combinatorics.perm_groups.PermutationGroup.is_alt_sym` (page 245)

Examples

```
>>> from sympy.combinatorics.util import _check_cycles_alt_sym
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12]])
>>> _check_cycles_alt_sym(a)
False
>>> b = Permutation([[0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10]])
>>> _check_cycles_alt_sym(b)
True
```

`sympy.combinatorics.util._distribute_gens_by_base(base, gens)`

Distribute the group elements `gens` by membership in basic stabilizers.

Notice that for a base $(b_1, b_2, \dots, b_k)$, the basic stabilizers are defined as $G^{(i)} = G_{b_1, \dots, b_{i-1}}$ for $i \in \{1, 2, \dots, k\}$.

Parameters “base” - a sequence of points in ‘{0, 1, ..., n-1}’

“gens” - a list of elements of a permutation group of degree ‘n’.

Returns List of length k , where k is

the length of base. The i -th entry contains those elements in

gens which fix the first i elements of base (so that the 0-th entry is equal to gens itself). If no element fixes the first i elements of base, the i -th element is set to a list containing the identity element.

See also:

[_strong_gens_from_distr](#) (page 286), [_orbits_transversals_from_bsgs](#) (page 284), [_handle_precomputed_bsgs](#) (page 283)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> D.strong_gens
[(0 1 2), (0 2), (1 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[[(0 1 2), (0 2), (1 2)],
  [(1 2)]]
```

```
sympy.combinatorics.util._handle_precomputed_bsgs(base, strong_gens,
                                                    transversals=None,
                                                    basic_orbits=None,
                                                    strong_gens_distr=None)
```

Calculate BSGS-related structures from those present.

The base and strong generating set must be provided; if any of the transversals, basic orbits or distributed strong generators are not provided, they will be calculated from the base and strong generating set.

Parameters “base” - the base

“strong_gens” - the strong generators

“transversals” - basic transversals

“basic_orbits” - basic orbits

“strong_gens_distr” - strong generators distributed by membership in basic

stabilizers

Returns (transversals, basic_orbits, strong_gens_distr) where transversals

are the basic transversals, basic_orbits are the basic orbits, and

strong_gens_distr are the strong generators distributed by membership in basic stabilizers.

See also:

[_orbits_transversals_from_bsgs](#) (page 284), [distribute_gens_by_base](#)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _handle_precomputed_bsgs
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> _handle_precomputed_bsgs(D.base, D.strong_gens,
... basic_orbits=D.basic_orbits)
([{0: (2), 1: (0 1 2), 2: (0 2)}, {1: (2), 2: (1 2)}], [[0, 1, 2], [1, 2]], [[(0,
→ 1 2), (0 2), (1 2)], [(1 2)]])
```

```
sympy.combinatorics.util._orbits_transversals_from_bsgs(base,
                                                         strong_gens_distr,
                                                         transver-
                                                         sals_only=False,
                                                         slp=False)
```

Compute basic orbits and transversals from a base and strong generating set.

The generators are provided as distributed across the basic stabilizers. If the optional argument `transversals_only` is set to `True`, only the transversals are returned.

Parameters “base” - the base

“strong_gens_distr” - strong generators distributed by membership in basic

stabilizers

“transversals_only” - a flag switching between returning only the transversals/ both orbits and transversals

“slp” - if “True”, return a list of dictionaries containing the

generator presentations of the elements of the transversals, i.e. the list of indices of generators from *strong_gens_distr*[*i*] such that their product is the relevant transversal element

See also:

[_distribute_gens_by_base](#) (page 282), [_handle_precomputed_bsgs](#) (page 283)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _orbits_transversals_from_bsgs
>>> from sympy.combinatorics.util import (_orbits_transversals_from_bsgs,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _orbits_transversals_from_bsgs(S.base, strong_gens_distr)
([[0, 1, 2], [1, 2]], [{0: (2), 1: (0 1 2), 2: (0 2 1)}, {1: (2), 2: (1 2)}])
```

```
sympy.combinatorics.util._remove_gens(base, strong_gens, basic_orbits=None,
                                       strong_gens_distr=None)
```

Remove redundant generators from a strong generating set.

Parameters “base” - a base

“strong_gens” - a strong generating set relative to “base”

“basic_orbits” - basic orbits

“strong_gens_distr” - strong generators distributed by membership in basic

stabilizers

Returns A strong generating set with respect to base which is a subset of strong_gens.

Notes

This procedure is outlined in [1],p.95.

References

[1] Holt, D., Eick, B., O’Brien, E. “Handbook of computational group theory”

Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.util import _remove_gens
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(15)
>>> base, strong_gens = S.schreier_sims_incremental()
>>> new_gens = _remove_gens(base, strong_gens)
>>> len(new_gens)
14
>>> _verify_bsgs(S, base, new_gens)
True
```

```
sympy.combinatorics.util._strip(g, base, orbits, transversals)
```

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

This is done by treating the sequence base as an actual base, and the orbits orbits and transversals transversals as basic orbits and transversals relative to it.

This process is called “sifting”. A sift is unsuccessful when a certain orbit element is not found or when after the sift the decomposition doesn’t end with the identity element.

The argument transversals is a list of dictionaries that provides transversal elements for the orbits orbits.

Parameters “g” - permutation to be decomposed

“base” - sequence of points

“orbits” - a list in which the “i”-th entry is an orbit of “base[i]” under some subgroup of the pointwise stabilizer of ‘

'base[0], base[1], ..., base[i - 1]'. The groups themselves are implicit in this function since the only information we need is encoded in the orbits and transversals

"transversals" - a list of orbit transversals associated with the orbits

"orbits".

See also:

`sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims` (page 254),
`sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims_random`
(page 256)

Notes

The algorithm is described in [1], pp.89-90. The reason for returning both the current state of the element being decomposed and the level at which the sifting ends is that they provide important information for the randomized version of the Schreier-Sims algorithm.

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.util import _strip
>>> S = SymmetricGroup(5)
>>> S.schreier_sims()
>>> g = Permutation([0, 2, 3, 1, 4])
>>> _strip(g, S.base, S.basic_orbits, S.basic_transversals)
((4), 5)
```

`sympy.combinatorics.util._strong_gens_from_distr(strong_gens_distr)`
Retrieve strong generating set from generators of basic stabilizers.

This is just the union of the generators of the first and second basic stabilizers.

Parameters "strong_gens_distr" - strong generators distributed by membership in basic stabilizers

See also:

`_distribute_gens_by_base` (page 282)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import (_strong_gens_from_distr,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> S.strong_gens
[(0 1 2), (2)(0 1), (1 2)]
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _strong_gens_from_distr(strong_gens_distr)
[(0 1 2), (2)(0 1), (1 2)]
```

Group constructors

`sympy.combinatorics.group_constructs.DirectProduct(*groups)`

Returns the direct product of several groups as a permutation group.

This is implemented much like the `__mul__` procedure for taking the direct product of two permutation groups, but the idea of shifting the generators is realized in the case of an arbitrary number of groups. A call to `DirectProduct(G1, G2, ..., Gn)` is generally expected to be faster than a call to `G1*G2*...*Gn` (and thus the need for this algorithm).

See also:

`__mul__`

Examples

```
>>> from sympy.combinatorics.group_constructs import DirectProduct
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

Test Utilities

`sympy.combinatorics.testutil._cmp_perm_lists(first, second)`

Compare two lists of permutations as sets.

This is used for testing purposes. Since the array form of a permutation is currently a list, `Permutation` is not hashable and cannot be put into a set.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _cmp_perm_lists
>>> a = Permutation([0, 2, 3, 4, 1])
```

(continues on next page)

(continued from previous page)

```
>>> b = Permutation([1, 2, 0, 4, 3])
>>> c = Permutation([3, 4, 0, 1, 2])
>>> ls1 = [a, b, c]
>>> ls2 = [b, c, a]
>>> _cmp_perm_lists(ls1, ls2)
True
```

`sympy.combinatorics.testutil._naive_list_centralizer(self, other, af=False)`

`sympy.combinatorics.testutil._verify_bsgs(group, base, gens)`

Verify the correctness of a base and strong generating set.

This is a naive implementation using the definition of a base and a strong generating set relative to it. There are other procedures for verifying a base and strong generating set, but this one will serve for more robust testing.

See also:

`sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims` (page 254)

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

`sympy.combinatorics.testutil._verify_centralizer(group, arg, centr=None)`

Verify the centralizer of a group/set/element inside another group.

This is used for testing `.centralizer()` from `sympy.combinatorics.perm_groups`

See also:

`_naive_list_centralizer` (page 288), `sympy.combinatorics.perm_groups.PermutationGroup.centralizer` (page 237), `_cmp_perm_lists` (page 287)

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _verify_centralizer
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])])
>>> _verify_centralizer(S, A, centr)
True
```

`sympy.combinatorics.testutil._verify_normal_closure(group, arg, closure=None)`

Tensor Canonicalization

`sympy.combinatorics.tensor_can.canonicalize(g, dummies, msym, *v)`
 canonicalize tensor formed by tensors

Parameters **g** : permutation representing the tensor

dummies : list representing the dummy indices

it can be a list of dummy indices of the same type or a list of lists of dummy indices, one list for each type of index; the dummy indices must come after the free indices, and put in order contravariant, covariant [d0, -d0, d1, -d1, ...]

msym : symmetry of the metric(s)

it can be an integer or a list; in the first case it is the symmetry of the dummy index metric; in the second case it is the list of the symmetries of the index metric for each type

v : list, (base_i, gens_i, n_i, sym_i) for tensors of type *i*

base_i, gens_i : BSGS for tensors of this type.

The BSGS should have minimal base under lexicographic ordering; if not, an attempt is made to get the minimal BSGS; in case of failure, `canonicalize_naive` is used, which is much slower.

n_i : number of tensors of type *i*.

sym_i : symmetry under exchange of component tensors of type *i*.

Both for msym and sym_i the cases are

- None no symmetry
- 0 commuting
- 1 anticommuting

Returns 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

Examples

one type of index with commuting metric;

A_{ab} and B_{ab} antisymmetric and commuting

$$T = A_{d_0 d_1} * B^{d_0}_{d_2} * B^{d_2 d_1}$$

$ord = [d_0, -d_0, d_1, -d_1, d_2, -d_2]$ order of the indices

$g = [1, 3, 0, 5, 4, 2, 6, 7]$

$T_c = 0$

```
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs, \
↳ canonicalize, bsgs_direct_product
>>> from sympy.combinatorics import Permutation
>>> base2a, gens2a = get_symmetric_group_sgs(2, 1)
>>> t0 = (base2a, gens2a, 1, 0)
>>> t1 = (base2a, gens2a, 2, 0)
>>> g = Permutation([1, 3, 0, 5, 4, 2, 6, 7])
```

(continues on next page)

(continued from previous page)

```
>>> canonicalize(g, range(6), 0, t0, t1)
0
```

same as above, but with B_{ab} anticommuting

$$T_c = -A^{d_0 d_1} * B_{d_0}^{d_2} * B_{d_1 d_2}$$

```
can = [0,2,1,4,3,5,7,6]
```

```
>>> t1 = (base2a, gens2a, 2, 1)
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 1, 4, 3, 5, 7, 6]
```

two types of indices $[a, b, c, d, e, f]$ and $[m, n]$, in this order, both with commuting metric f^{abc} antisymmetric, commuting

A_{ma} no symmetry, commuting

$$T = f^c_{da} * f^f_{eb} * A_m^d * A^{mb} * A_n^a * A^{ne}$$

```
ord = [c,f,a,-a,b,-b,d,-d,e,-e,m,-m,n,-n]
```

```
g = [0,7,3, 1,9,5, 11,6, 10,4, 13,2, 12,8, 14,15]
```

The canonical tensor is $T_c = -f^{cab} * f^{fde} * A^m_a * A_{md} * A^n_b * A_{ne}$

```
can = [0,2,4, 1,6,8, 10,3, 11,7, 12,5, 13,9, 15,14]
```

```
>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base_l, gens_l = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base_l, gens_l, base_l, gens_l)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_A, gens_A, 4, 0)
>>> dummies = [range(2, 10), range(10, 14)]
>>> g = Permutation([0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15])
>>> canonicalize(g, dummies, [0, 0], t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

Algorithm

First one uses `canonical_free` to get the minimum tensor under lexicographic order, using only the slot symmetries. If the component tensors have not minimal BSGS, it is attempted to find it; if the attempt fails `canonicalize_naive` is used instead.

Compute the residual slot symmetry keeping fixed the free indices using `tensor_gens(base, gens, list_free_indices, sym)`.

Reduce the problem eliminating the free indices.

Then use `double_coset_can_rep` and lift back the result reintroducing the free indices.

```
sympy.combinatorics.tensor_can.double_coset_can_rep(dummies, sym, b_S,
                                                    sgens, S_transversals, g)
```

Butler-Portugal algorithm for tensor canonicalization with dummy indices

dummies list of lists of dummy indices, one list for each type of index; the dummy indices are put in order contravariant, covariant $[d_0, -d_0, d_1, -d_1, \dots]$.

sym list of the symmetries of the index metric for each type.

possible symmetries of the metrics

- 0 symmetric
- 1 antisymmetric
- None no symmetry

b_S base of a minimal slot symmetry BSGS.

sgens generators of the slot symmetry BSGS.

S_transversals transversals for the slot BSGS.

g permutation representing the tensor.

Return 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

A tensor with dummy indices can be represented in a number of equivalent ways which typically grows exponentially with the number of indices. To be able to establish if two tensors with many indices are equal becomes computationally very slow in absence of an efficient algorithm.

The Butler-Portugal algorithm [3] is an efficient algorithm to put tensors in canonical form, solving the above problem.

Portugal observed that a tensor can be represented by a permutation, and that the class of tensors equivalent to it under slot and dummy symmetries is equivalent to the double coset $D * g * S$ (Note: in this documentation we use the conventions for multiplication of permutations p, q with $(p*q)(i) = p[q[i]]$ which is opposite to the one used in the Permutation class)

Using the algorithm by Butler to find a representative of the double coset one can find a canonical form for the tensor.

To see this correspondence, let g be a permutation in array form; a tensor with indices ind (the indices including both the contravariant and the covariant ones) can be written as

$$t = T(ind[g[0], ..., ind[g[n - 1]]],$$

where $n = len(ind)$; g has size $n + 2$, the last two indices for the sign of the tensor (trick introduced in [4]).

A slot symmetry transformation s is a permutation acting on the slots $t \rightarrow T(ind[(g * s)[0]], ..., ind[(g * s)[n - 1]])$

A dummy symmetry transformation acts on ind $t \rightarrow T(ind[(d * g)[0]], ..., ind[(d * g)[n - 1]])$

Being interested only in the transformations of the tensor under these symmetries, one can represent the tensor by g , which transforms as

$g \rightarrow d * g * s$, so it belongs to the coset $D * g * S$.

Let us explain the conventions by an example.

Given a tensor T^{d3d2d1}_{d1d2d3} **with the slot symmetries** $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$

$$T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$$

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering $d1, d2, d3$ then and contravariant index before covariant index; that is the canonical form of the tensor.

The canonical form is $-T^{d1d2d3}_{d1d2d3}$ obtained using $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$.

To convert this problem in the input for this function, use the following labelling of the index names (- for covariant for short) $d1, -d1, d2, -d2, d3, -d3$

T^{d3d2d1}_{d1d2d3} corresponds to $g = [4, 2, 0, 1, 3, 5, 6, 7]$ where the last two indices are for the sign
 $sgens = [Permutation(0, 2)(6, 7), Permutation(0, 4)(6, 7)]$

$sgens[0]$ is the slot symmetry $-(0, 2)$ $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$

$sgens[1]$ is the slot symmetry $-(0, 4)$ $T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$

The dummy symmetry group D is generated by the strong base generators $[(0, 1), (2, 3), (4, 5), (0, 1)(2, 3), (2, 3)(4, 5)]$

The dummy symmetry acts from the left $d = [1, 0, 2, 3, 4, 5, 6, 7]$ exchange $d1- > -d1$
 $T^{d3d2d1}_{d1d2d3} == T^{d3d2}_{d1}{}^{d1}_{d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] - > [4, 2, 1, 0, 3, 5, 6, 7] =_a f_r mul(d, g)$ which differs from $_a f_r mul(g, d)$.

The slot symmetry acts from the right $s = [2, 1, 0, 3, 4, 5, 7, 6]$ exchanges slots 0 and 2 and changes sign $T^{d3d2d1}_{d1d2d3} == -T^{d1d2d3}_{d1d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] - > [0, 2, 4, 1, 3, 5, 7, 6] =_a f_r mul(g, s)$

Example in which the tensor is zero, same slot symmetries as above: $T^{d3}_{d1, d2}{}^{d1}_{d3}{}^{d2}$

$= -T^{d3}_{d1, d3}{}^{d1}_{d2}{}^{d2}$ under slot symmetry $-(2, 4)$;

$= T^{d3d1}_{d3d1}{}^{d3d1}_{d2}{}^{d2}$ under slot symmetry $-(0, 2)$;

$= T^{d3}_{d1d3}{}^{d1}_{d2}{}^{d2}$ symmetric metric;

$= 0$ since two of these lines have tensors differ only for the sign.

The double coset $D * g * S$ consists of permutations $h = d * g * s$ corresponding to equivalent tensors; if there are two h which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to $[d1, -d1, d2, -d2, d3, -d3]$ that is $rep = \min(D * g * S) = \min([d * g * s \text{ for } d \text{ in } D \text{ for } s \text{ in } S])$

The indices are fixed one by one; first choose the lowest index for slot 0, then the lowest remaining index for slot 1, etc. Doing this one obtains a chain of stabilizers

$S- > S_{b0-} > S_{b0, b1-} > \dots$ and $D- > D_{p0-} > D_{p0, p1-} > \dots$

where $[b0, b1, \dots] = \text{range}(b)$ is a base of the symmetric group; the strong base b_S of S is an ordered sublist of it; therefore it is sufficient to compute once the strong base generators of S using the Schreier-Sims algorithm; the stabilizers of the strong base generators are the strong base generators of the stabilizer subgroup.

$dbase = [p0, p1, \dots]$ is not in general in lexicographic order, so that one must recompute the strong base generators each time; however this is trivial, there is no need to use the Schreier-Sims algorithm for D .

The algorithm keeps a TAB of elements (s_i, d_i, h_i) where $h_i = d_i * g * s_i$ satisfying $h_i[j] = p_j$ for $0 \leq j < i$ starting from $s_0 = id, d_0 = id, h_0 = g$.

The equations $h_0[0] = p_0, h_1[1] = p_1, \dots$ are solved in this order, choosing each time the lowest possible value of p_i

For $j < i$ $d_i * g * s_i * S_{b0, \dots, b_{i-1}} * b_j = D_{p0, \dots, p_{i-1}} * p_j$ so that for dx in $D_{p0, \dots, p_{i-1}}$ and sx in $S_{base[0], \dots, base[i-1]}$ one has $dx * d_i * g * s_i * sx * b_j = p_j$

Search for dx, sx such that this equation holds for $j = i$; it can be written as $s_i * sx * b_j = J, dx * d_i * g * J = p_j$ $sx * b_j = s_i * ** - 1 * J$; $sx = \text{trace}(s_i * ** - 1, S_{b0, \dots, b_{i-1}})$ $dx * ** - 1 * p_j = d_i * g * J$; $dx = \text{trace}(d_i * g * J, D_{p0, \dots, p_{i-1}})$

$s_{i+1} = s_i * \text{trace}(s_i * ** - 1 * J, S_{b0, \dots, b_{i-1}})$ $d_{i+1} = \text{trace}(d_i * g * J, D_{p0, \dots, p_{i-1}}) * ** - 1 * d_i$ $h_{i+1} * b_i = d_{i+1} * g * s_{i+1} * b_i = p_i$

$h_n * b_j = p_j$ for all j , so that h_n is the solution.

Add the found (s, d, h) to TAB1.

At the end of the iteration sort TAB1 with respect to the h ; if there are two consecutive h in TAB1 which differ only for the sign, the tensor is zero, so return 0; if there are two consecutive h which are equal, keep only one.

Then stabilize the slot generators under i and the dummy generators under p_i .

Assign $TAB = TAB1$ at the end of the iteration step.

At the end TAB contains a unique (s, d, h) , since all the slots of the tensor h have been fixed to have the minimum value according to the symmetries. The algorithm returns h .

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an i which does not belong to the slot base for which p_i is fixed by the dummy symmetry only, while i is not invariant from the slot stabilizer, so p_i is not in general the minimal value.

This algorithm differs slightly from the original algorithm [3]: the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors h are eliminated from TAB.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.tensor_can import double_coset_can_rep, get_
↳ transversals
>>> gens = [Permutation(x) for x in [[2, 1, 0, 3, 4, 5, 7, 6], [4, 1, 2, 3, 0, 5, 7, 6]]]
>>> base = [0, 2]
>>> g = Permutation([4, 2, 0, 1, 3, 5, 6, 7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
[0, 1, 2, 3, 4, 5, 7, 6]
```

```
>>> g = Permutation([4, 1, 3, 0, 5, 2, 6, 7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
0
```

`sympy.combinatorics.tensor_can.get_symmetric_group_sgs(n, antisym=False)`

Return base, gens of the minimal BSGS for (anti)symmetric tensor

n rank of the tensor

`antisym = False` symmetric tensor `antisym = True` antisymmetric tensor

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs
>>> Permutation.print_cyclic = True
>>> get_symmetric_group_sgs(3)
([0, 1], [(4)(0 1), (4)(1 2)])
```

```
sympy.combinatorics.tensor_can.bsgs_direct_product(base1, gens1, base2,
                                                    gens2, signed=True)
```

direct product of two BSGS

base1 base of the first BSGS.

gens1 strong generating sequence of the first BSGS.

base2, gens2 similarly for the second BSGS.

signed flag for signed permutations.

Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.tensor_can import (get_symmetric_group_sgs, bsgs_
↳direct_product)
>>> Permutation.print_cyclic = True
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base2, gens2 = get_symmetric_group_sgs(2)
>>> bsgs_direct_product(base1, gens1, base2, gens2)
([1], [(4)(1 2)])
```

Finitely Presented Groups

Introduction

This module presents the functionality designed for computing with finitely-presented groups (fp-groups for short). The name of the corresponding SymPy object is `FpGroup`. The functions or classes described here are studied under **computational group theory**. All code examples assume:

```
>>> from sympy.combinatorics.free_groups import free_group, vfree_group, xfree_group
>>> from sympy.combinatorics.fp_groups import FpGroup, CosetTable, coset_enumeration_r
```

Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings

- The construction of fp-groups using a free group and a list of words in generators of that free group.
- Index determination using the famous Todd-Coxeter procedure.
- The construction of all subgroups having index less than some (small) specified positive integer, using the Low-Index Subgroups algorithm.
- Algorithms for computing presentations of a subgroup of finite index in a group defined by finite presentation.

For a description of fundamental algorithms of finitely presented groups we often make use of Handbook of Computational Group Theory.

The Construction of Finitely Presented Groups

Finitely presented groups are constructed by factoring a free group by a set of relators. The set of relators is taken in as a list of words in generators of free group in SymPy, using a list provides ordering to the relators. If the list of relators is empty, the associated free group is returned.

Example of construction of a finitely-presented group. The symmetric group of degree 4 may be represented as a two generator group with presentation $\langle a, b \mid a^2, b^3, (ab)^4 \rangle$. Giving the relations as a list of relators, group in SymPy would be specified as:

```
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> G
<fp group on the generators (a, b)>
```

Currently groups with relators having presentation like $\langle r, s, t \mid r^2, s^2, t^2, rst = str = trs \rangle$ will have to be specified as:

```
>>> F, r, s, t = free_group("r, s, t")
>>> G = FpGroup(F, [r**2, s**2, t**2, r*s*t*r**-1*t**-1*s**-1, s*t*r*s**-1*r**-1*t**-1])
```

Obviously this is not a unique way to make that particular group, but the point is that in case of equality with non-identity the user has to manually do that.

Free Groups and Words

Construction of a Free Group

`free_group("gen0, gen1, ..., gen_(n-1)")` constructs a free group F on n generators, where n is a positive integer. The i -th generator of F may be obtained using the method `.generators[i]`, $i = 0, \dots, n-1$.

```
>>> F, x, y = free_group("x, y")
```

creates a free group F of rank 2 and assigns the variables x and y to the two generators.

```
>>> F = vfree_group("x, y")
>>> F
<free group on the generators (x, y)>
```

creates a free group F of rank 2, with tuple of generators `F.generators`, and inserts x and y as generators into the global namespace.

```
>>> F = xfree_group("x, y")
>>> F
(<free group on the generators (x, y)>, (x, y))
>>> x**2
x**2
```

creates a free groups `F[0]` of rank 2, with tuple of generators `F[1]`.

Construction of words

This section is applicable to words of `FreeGroup` as well as `FpGroup`. When we say word in SymPy, it actually means a **reduced word**, since the words are automatically reduced. Given a group G defined on n generators $x_1, x_2, x_3, \dots, x_n$, a word is constructed as $s_1^{r_1} s_2^{r_2} \dots s_k^{r_k}$ where $s_i \in \{x_1, x_2, \dots, x_n\}$, $r_i \in \mathbb{Z}$ for all k .

Each word can be constructed in a variety of ways, since after reduction they may be equivalent.

Coset Enumeration: The Todd-Coxeter Algorithm

This section describes the use of coset enumeration techniques in SymPy. The algorithm used for coset enumeration procedure is Todd-Coxeter algorithm and is developed in SymPy using [Ho05] and [CDHW73]. The reader should consult [CDHW73] and [Hav91] for a general description of the algorithm.

We have two strategies of coset enumeration relator-based and coset-table based and the two have been implemented as `coset_enumeration_r`, `coset_enumeration_c` respectively. The two strategies differ in the way they make new definitions for the cosets.

Though from the user point of view it is suggested to rather use the `.coset_enumeration` method of `FpGroup` and specify the `strategy` argument.

strategy: (default="relator_based") specifies the strategy of coset enumeration to be used, possible values are "relator_based" or "coset_table_based".

CosetTable

Class used to manipulate the information regarding the coset enumeration of the finitely presented group G on the cosets of the subgroup H .

Basically a coset table `CosetTable(G,H)`, is the permutation representation of the finitely presented group on the cosets of a subgroup. Most of the set theoretic and group functions use the regular representation of G , i.e., the coset table of G over the trivial subgroup.

The actual mathematical coset table is obtained using `.table` attribute and is a list of lists. For each generator g of G it contains a column and the next column corresponds to g^{-1} and so on for other generators, so in total it has $2 * G.\text{rank}()$ columns. Each column is simply a list of integers. If l is the generator list for the generator g and if $l[i] = j$ then generator g takes the coset i to the coset j by multiplication from the right.

For finitely presented groups, a coset table is computed by a Todd-Coxeter coset enumeration. Note that you may influence the performance of that enumeration by changing the values of the variable `CosetTable.coset_table_max_limit`.

Attributes of CosetTable

For `CosetTable(G, H)` where G is the group and H is the subgroup.

- `n`: A non-negative integer, non-mutable attribute, dependently calculated as the maximum among the live-cosets (i.e. Ω).
- `table`: A list of lists, mutable attribute, mathematically represents the coset table.

- **omega**: A list, dependent on the internal attribute `p`. Ω represents the list of live-cosets. A standard coset-table has its $\Omega = \{0, 1, \dots, index - 1\}$ where *index* is the index of subgroup *H* in *G*.

For experienced users we have a number of parameters that can be used to manipulate the algorithm, like

- **coset_table_max_limit** (default value = 4096000): manipulate the maximum number of cosets allowed in coset enumeration, i.e. the number of rows allowed in coset table. A coset enumeration will not finish if the subgroup does not have finite index, and even if it has it may take many more intermediate cosets than the actual index of the subgroup is. To avoid a coset enumeration “running away” therefore SymPy has a “safety stop” built-in. This is controlled by this variable. For example:

```
>>> CosetTable.coset_table_max_limit = 50
>>> F, a, b = free_group("a, b")
>>> Cox = FpGroup(F, [a**6, b**6, (a*b)**2, (a**2*b**2)**2, (a**3*b**3)**5])
>>> C_r = coset_enumeration_r(Cox, [a])
Traceback (most recent call last):
...
ValueError: the coset enumeration has defined more than 50 cosets
```

- **max_stack_size** (default value = 500): manipulate the maximum size of deduction_stack above or equal to which the stack is emptied.

Compression and Standardization

For any two entries i, j with $i < j$ in coset table, the first occurrence of i in a coset table precedes the first occurrence of j with respect to the usual row-wise ordering of the table entries. We call such a table a standard coset table. To standardize a `CosetTable` we use the `.standardize` method.

Note the method alters the given table, it does not create a copy.

Subgroups of Finite Index

The functionality in this section are concerned with the construction of subgroups of finite index. We describe a method for computing all subgroups whose index does not exceed some (modest) integer bound.

Low Index Subgroups

low_index_subgroups(*G*, *N*): Given a finitely presented group $G = \langle X \mid R \rangle$ (can be a free group), and *N* a positive integer, determine the conjugacy classes of subgroups of *G* whose indices is less than or equal to *N*.

For example to find all subgroups of $G = \langle a, b \mid a^2 = b^3 = (ab)^4 = 1 \rangle$ having index ≤ 4 , can be found as follows:

```
>>> from sympy.combinatorics.fp_groups import low_index_subgroups
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> l = low_index_subgroups(G, 4)
>>> for coset_table in l:
```

(continues on next page)

(continued from previous page)

```
...     print(coset_table.table)
...
[[0, 0, 0, 0]]
[[0, 0, 1, 2], [1, 1, 2, 0], [3, 3, 0, 1], [2, 2, 3, 3]]
[[0, 0, 1, 2], [2, 2, 2, 0], [1, 1, 0, 1]]
[[1, 1, 0, 0], [0, 0, 1, 1]]
```

This returns the coset tables of subgroups of satisfying the property that index, *index*, of subgroup in group is $\leq n$.

Constructing a presentation for a subgroup

In this section we discuss finding the presentation of a subgroup in a finitely presentation group. While the subgroup is currently allowed as input only in the form of a list of generators for the subgroup, you can expect the functionality of a coset table as input for subgroup in the group in near future.

There are two ways to construct a set of defining relations for subgroup from those of G . First is on a set of Schreier generators, known generally as Reidemeister-Schreier algorithm or on the given list of generators of H .

Reidemeister Schreier algorithm

called using `reidemeister_presentation(G, Y)` where G is the group and Y is a list of generators for subgroup H whose presentation we want to find.

```
>>> from sympy.combinatorics.fp_groups import reidemeister_presentation
>>> F, x, y = free_group("x, y")
>>> f = FpGroup(F, [x**3, y**5, (x*y)**2])
>>> H = [x*y, x**-1*y**-1*x*y*x]
>>> p1 = reidemeister_presentation(f, H)
>>> p1
((y_1, y_2), (y_1**2, y_2**3, y_2*y_1*y_2*y_1*y_2*y_1))
```

Bibliography

5.3 Number Theory

5.3.1 Ntheory Class Reference

class `sympy.ntheory.generate.Sieve`

An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes. When a lookup is requested involving an odd number that has not been sieved, the sieve is automatically extended up to that number.

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> 25 in sieve
False
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

extend(n)

Grow the sieve to cover all primes $\leq n$ (a real number).

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend(30)
>>> sieve[10] == 29
True
```

extend_to_no(i)

Extend to include the i th prime number.

i must be an integer.

The list is extended by 50% if it is too short, so it is likely that it will be longer than requested.

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend_to_no(9)
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

mobiusrange(a, b)

Generate all mobius numbers for the range $[a, b)$.

Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.mobiusrange(7, 18)])
[-1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1]
```

primerange(a, b)

Generate all prime numbers in the range $[a, b)$.

Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.primerange(7, 18)])
[7, 11, 13, 17]
```

search(n)

Return the indices i, j of the primes that bound n .

If n is prime then $i == j$.

Although n can be an expression, if ceiling cannot convert it to an integer then an error will be raised.

Examples

```
>>> from sympy import sieve
>>> sieve.search(25)
(9, 10)
>>> sieve.search(23)
(9, 9)
```

totientrange(a, b)

Generate all totient numbers for the range $[a, b)$.

Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.totientrange(7, 18)])
[6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16]
```

5.3.2 Ntheory Functions Reference

sympy.ntheory.generate.prime(nth)

Return the n th prime, with the primes indexed as $\text{prime}(1) = 2$, $\text{prime}(2) = 3$, etc.... The n th prime is approximately $n \cdot \log(n)$.

Logarithmic integral of x is a pretty nice approximation for number of primes $\leq x$, i.e. $\text{li}(x) \sim \pi(x)$ In fact, for the numbers we are concerned about ($x < 1e11$), $\text{li}(x) - \pi(x) < 50000$

Also, $\text{li}(x) > \pi(x)$ can be safely assumed for the numbers which can be evaluated by this function.

Here, we find the least integer m such that $\text{li}(m) > n$ using binary search. Now $\pi(m-1) < \text{li}(m-1) \leq n$,

We find $\pi(m - 1)$ using `primepi` function.

Starting from m , we have to find $n - \pi(m-1)$ more primes.

For the inputs this implementation can handle, we will have to test primality for at max about 10^{*5} numbers, to get our answer.

See also:

`sympy.ntheory.primetest.isprime` (page 326) Test if n is prime

`primerange` (page 302) Generate all primes in a given range

`primepi` (page 301) Return the number of primes less than or equal to n

References

- https://en.wikipedia.org/wiki/Prime_number_theorem#Table_of_.CF.80.28x.29.2C_.2F_log_x.2C_and_li.28x.29
- https://en.wikipedia.org/wiki/Prime_number_theorem#Approximations_for_the_nth_prime_number
- https://en.wikipedia.org/wiki/Skewes%27_number

Examples

```
>>> from sympy import prime
>>> prime(10)
29
>>> prime(1)
2
>>> prime(100000)
1299709
```

`sympy.ntheory.generate.primepi(n)`

Return the value of the prime counting function $\pi(n)$ = the number of prime numbers less than or equal to n .

Algorithm Description:

In sieve method, we remove all multiples of prime p except p itself.

Let $\phi(i, j)$ be the number of integers $2 \leq k \leq i$ which remain after sieving from primes less than or equal to j . Clearly, $\pi(n) = \phi(n, \sqrt{n})$

If j is not a prime, $\phi(i, j) = \phi(i, j - 1)$

if j is a prime, We remove all numbers(except j) whose smallest prime factor is j .

Let $x = j \cdot a$ be such a number, where $2 \leq a \leq i / j$ Now, after sieving from primes $\leq j - 1$, a must remain (because x , and hence a has no prime factor $\leq j - 1$) Clearly, there are $\phi(i / j, j - 1)$ such a which remain on sieving from primes $\leq j - 1$

Now, if a is a prime less than equal to $j - 1$, $x = j \cdot a$ has smallest prime factor = a , and has already been removed(by sieving from a). So, we don't need to remove it again. (Note: there will be $\pi(j - 1)$ such x)

Thus, number of x , that will be removed are: $\phi(i / j, j - 1) - \phi(j - 1, j - 1)$ (Note that $\pi(j - 1) = \phi(j - 1, j - 1)$)

$\Rightarrow \phi(i, j) = \phi(i, j - 1) - \phi(i / j, j - 1) + \phi(j - 1, j - 1)$

So, following recursion is used and implemented as dp:

$\phi(a, b) = \phi(a, b - 1)$, if b is not a prime
 $\phi(a, b) = \phi(a, b - 1) - \phi(a / b, b - 1) + \phi(b - 1, b - 1)$, if b is prime

Clearly a is always of the form $\text{floor}(n / k)$, which can take at most $2 \cdot \sqrt{n}$ values. Two arrays `arr1`, `arr2` are maintained `arr1[i] =  $\phi(i, j)$` , `arr2[i] =  $\phi(n // i, j)$`

Finally the answer is `arr2[1]`

See also:

`sympy.ntheory.primetest.isprime` (page 326) Test if `n` is prime

`primerange` (page 302) Generate all primes in a given range

`prime` (page 300) Return the `nth` prime

Examples

```
>>> from sympy import primepi
>>> primepi(25)
9
```

`sympy.ntheory.generate.nextprime(n, ith=1)`

Return the `ith` prime greater than `n`.

`i` must be an integer.

See also:

`prevprime` (page 302) Return the largest prime smaller than `n`

`primerange` (page 302) Generate all primes in a given range

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> from sympy import nextprime
>>> [(i, nextprime(i)) for i in range(10, 15)]
[(10, 11), (11, 13), (12, 13), (13, 17), (14, 17)]
>>> nextprime(2, ith=2) # the 2nd prime after 2
5
```

`sympy.ntheory.generate.prevprime(n)`

Return the largest prime smaller than `n`.

See also:

`nextprime` (page 302) Return the `ith` prime greater than `n`

`primerange` (page 302) Generates all primes in a given range

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> from sympy import prevprime
>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

`sympy.ntheory.generate.primerange(a, b)`

Generate a list of all prime numbers in the range $[a, b)$.

If the range exists in the default sieve, the values will be returned from there; otherwise values will be returned but will not modify the sieve.

See also:

nextprime (page 302) Return the i th prime greater than n

prevprime (page 302) Return the largest prime smaller than n

randprime (page 303) Returns a random prime in a given range

primorial (page 304) Returns the product of primes based on condition

Sieve.primerange (page 299) return range from already computed primes or extend the sieve to contain the requested range.

Notes

Some famous conjectures about the occurrence of primes in a given range are [1]:

- **Twin primes: though often not, the following will give 2 primes an infinite number of times:** `primerange(6*n - 1, 6*n + 2)`
- **Legendre's: the following always yields at least one prime** `primerange(n**2, (n+1)**2+1)`
- **Bertrand's (proven): there is always a prime in the range** `primerange(n, 2*n)`
- **Brocard's: there are at least four primes in the range** `primerange(prime(n)**2, prime(n+1)**2)`

The average gap between primes is $\log(n)$ [2]; the gap between primes can be arbitrarily large since sequences of composite numbers are arbitrarily large, e.g. the numbers in the sequence $n! + 2, n! + 3 \dots n! + n$ are all composite.

References

1. http://en.wikipedia.org/wiki/Prime_number
2. <http://primes.utm.edu/notes/gaps.html>

Examples

```
>>> from sympy import primerange, sieve
>>> print([i for i in primerange(1, 30)])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The Sieve method, `primerange`, is generally faster but it will occupy more memory as the sieve stores values. The default instance of Sieve, named `sieve`, can be used:

```
>>> list(sieve.primerange(1, 30))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

`sympy.ntheory.generate.randprime(a, b)`

Return a random prime number in the range $[a, b)$.

Bertrand's postulate assures that `randprime(a, 2*a)` will always succeed for $a > 1$.

See also:

primerange (page 302) Generate all primes in a given range

References

- http://en.wikipedia.org/wiki/Bertrand's_postulate

Examples

```
>>> from sympy import randprime, isprime
>>> randprime(1, 30)
13
>>> isprime(randprime(1, 30))
True
```

`sympy.ntheory.generate.primorial(n, nth=True)`

Returns the product of the first n primes (default) or the primes less than or equal to n (when `nth=False`).

```
>>> from sympy.ntheory.generate import primorial, randprime, primerange
>>> from sympy import factorint, Mul, primefactors, sqrt
>>> primorial(4) # the first 4 primes are 2, 3, 5, 7
210
>>> primorial(4, nth=False) # primes <= 4 are 2 and 3
6
>>> primorial(1)
2
>>> primorial(1, nth=False)
1
>>> primorial(sqrt(101), nth=False)
210
```

One can argue that the primes are infinite since if you take a set of primes and multiply them together (e.g. the primorial) and then add or subtract 1, the result cannot be divided by any of the original factors, hence either 1 or more new primes must divide this product of primes.

In this case, the number itself is a new prime:

```
>>> factorint(primorial(4) + 1)
{211: 1}
```

In this case two new primes are the factors:

```
>>> factorint(primorial(4) - 1)
{11: 1, 19: 1}
```

Here, some primes smaller and larger than the primes multiplied together are obtained:

```
>>> p = list(primerange(10, 20))
>>> sorted(set(primefactors(Mul(*p) + 1)).difference(set(p)))
[2, 5, 31, 149]
```

See also:

primerange (page 302) Generate all primes in a given range

`sympy.ntheory.generate.cycle_length(f, x0, nmax=None, values=False)`

For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if `values` is `True` then the terms of the sequence will be returned instead. The sequence is started with value `x0`.

Note: more than the first `lambda + mu` terms may be returned and this is the cost of cycle detection with Brent's method; there are, however, generally less terms calculated than would have been calculated if the proper ending point were determined, e.g. by using Floyd's method.

```
>>> from sympy.ntheory.generate import cycle_length
```

This will yield successive values of `i <- func(i)`:

```
>>> def iter(func, i):
...     while 1:
...         ii = func(i)
...         yield ii
...         i = ii
... 
```

A function is defined:

```
>>> func = lambda i: (i**2 + 1) % 51
```

and given a seed of 4 and the mu and lambda terms calculated:

```
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```
>>> n = cycle_length(func, 4, values=True)
>>> list(ni for ni in n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, `nmax` can be used to exit early (and `mu` will be returned as `None`):

```
>>> next(cycle_length(func, 4, nmax = 4))
(4, None)
>>> [ni for ni in cycle_length(func, 4, nmax = 4, values=True)]
[17, 35, 2, 5]
```

Code modified from: http://en.wikipedia.org/wiki/Cycle_detection.

`sympy.ntheory.generate.composite(nth)`

Return the *nth* composite number, with the composite numbers indexed as `composite(1) = 4`, `composite(2) = 6`, etc....

See also:

`sympy.ntheory.primetest.isprime` (page 326) Test if *n* is prime

`primerange` (page 302) Generate all primes in a given range

`primepi` (page 301) Return the number of primes less than or equal to *n*

`prime` (page 300) Return the *nth* prime

`compositepi` (page 306) Return the number of positive composite numbers less than or equal to *n*

Examples

```
>>> from sympy import composite
>>> composite(36)
52
>>> composite(1)
4
>>> composite(17737)
20000
```

`sympy.ntheory.generate.compositepi(n)`

Return the number of positive composite numbers less than or equal to *n*. The first positive composite is 4, i.e. `compositepi(4) = 1`.

See also:

`sympy.ntheory.primetest.isprime` (page 326) Test if *n* is prime

`primerange` (page 302) Generate all primes in a given range

`prime` (page 300) Return the *nth* prime

`primepi` (page 301) Return the number of primes less than or equal to *n*

`composite` (page 305) Return the *nth* composite number

Examples

```
>>> from sympy import compositepi
>>> compositepi(25)
15
>>> compositepi(1000)
831
```

`sympy.ntheory.factor_.smoothness(n)`

Return the B-smooth and B-power smooth values of *n*.

The smoothness of *n* is the largest prime factor of *n*; the power-smoothness is the largest divisor raised to its multiplicity.


```
>>> from sympy.ntheory.factor_ import smoothness
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

See also:

[factorint](#) (page 312), [smoothness_p](#) (page 307)

`sympy.ntheory.factor_.smoothness_p(n, m=-1, power=0, visual=None)`

Return a list of `[m, (p, (M, sm(p + m), psm(p + m)))]` where:

1. p^*M is the base- p divisor of n
2. $sm(p + m)$ is the smoothness of $p + m$ ($m = -1$ by default)
3. $psm(p + m)$ is the power smoothness of $p + m$

The list is sorted according to smoothness (default) or by power smoothness if `power=1`.

The smoothness of the numbers to the left ($m = -1$) or right ($m = 1$) of a factor govern the results that are obtained from the $p \pm 1$ type factoring methods.

```
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> smoothness_p(10431, m=1)
(1, [(3, (2, 2, 4)), (19, (1, 5, 5)), (61, (1, 31, 31))])
>>> smoothness_p(10431)
(-1, [(3, (2, 2, 2)), (19, (1, 3, 9)), (61, (1, 5, 5))])
>>> smoothness_p(10431, power=1)
(-1, [(3, (2, 2, 2)), (61, (1, 5, 5)), (19, (1, 3, 9))])
```

If `visual=True` then an annotated string will be returned:

```
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

This string can also be generated directly from a factorization dictionary and vice versa:

```
>>> factorint(17*9)
{3: 2, 17: 1}
>>> smoothness_p(_)
'p**i=3**2 has p-1 B=2, B-pow=2\np**i=17**1 has p-1 B=2, B-pow=16'
>>> smoothness_p(_)
{3: 2, 17: 1}
```

The table of the output logic is:

	Visual		
Input	True	False	other
dict	str	tuple	str
str	str	tuple	dict
tuple	str	tuple	str
n	str	tuple	tuple
mul	str	tuple	tuple

See also:

`factorint` (page 312), `smoothness` (page 306)

`sympy.ntheory.factor_.trailing(n)`

Count the number of trailing zero digits in the binary representation of n , i.e. determine the largest power of 2 that divides n .

Examples

```
>>> from sympy import trailing
>>> trailing(128)
7
>>> trailing(63)
0
```

`sympy.ntheory.factor_.multiplicity(p, n)`

Find the greatest integer m such that p^m divides n .

Examples

```
>>> from sympy.ntheory import multiplicity
>>> from sympy.core.numbers import Rational as R
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, R(1, 9))
-2
```

`sympy.ntheory.factor_.perfect_power(n, candidates=None, big=True, factor=True)`

Return (b, e) such that $n == b^e$ if n is a perfect power; otherwise return `False`.

By default, the base is recursively decomposed and the exponents collected so the largest possible e is sought. If `big=False` then the smallest possible e (thus prime) will be chosen.

If `candidates` for exponents are given, they are assumed to be sorted and the first one that is larger than the computed maximum will signal failure for the routine.

If `factor=True` then simultaneous factorization of n is attempted since finding a factor indicates the only possible root for n . This is `True` by default since only a few small factors will be tested in the course of searching for the perfect power.

Examples

```
>>> from sympy import perfect_power
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big = False)
(4, 2)
```

`sympy.ntheory.factor_.pollard_rho(n, s=2, a=1, retries=5, seed=1234, max_steps=None, F=None)`

Use Pollard's rho method to try to extract a nontrivial factor of n . The returned factor may be a composite number. If no factor is found, `None` is returned.

The algorithm generates pseudo-random values of x with a generator function, replacing x with $F(x)$. If F is not supplied then the function $x^2 + a$ is used. The first value supplied to $F(x)$ is s . Upon failure (if `retries` is > 0) a new a and s will be supplied; the a will be ignored if F was supplied.

The sequence of numbers generated by such functions generally have a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 – this leader and loop look a bit like the Greek letter rho, and thus the name, ‘rho’.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> from sympy.ntheory.generate import cycle_length
>>> n = 16843009
>>> F = lambda x: (2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     print('loop length = %4i; leader length = %3i' % next(cycle_length(F, s)))
...
loop length = 2489; leader length = 42
loop length = 78; leader length = 120
loop length = 1482; leader length = 99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x=2
>>> for i in range(9):
...     x=(x**2+12)%17
...     print(x)
...
16
13
11
14
4
11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12)%17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12)%17, 2, values=True))
[16, 13, 11, 14, 4]
```

Instead of checking the differences of all generated values for a gcd with n , only the k th and $2*k$ th numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If `max_steps` is specified, the iteration is cancelled with a failure after the specified number of steps.

References

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 229-231

Examples

```
>>> from sympy import pollard_rho
>>> n=16843009
>>> F=lambda x:(2048*pow(x,2,n) + 32767) % n
>>> pollard_rho(n, F=F)
257
```

Use the default setting with a bad value of a and no retries:

```
>>> pollard_rho(n, a=n-2, retries=0)
```

If `retries` is > 0 then perhaps the problem will correct itself when new values are generated for a :

```
>>> pollard_rho(n, a=n-2, retries=1)
257
```

`sympy.ntheory.factor_.pollard_pm1(n, B=10, a=2, retries=0, seed=1234)`

Use Pollard's $p-1$ method to try to extract a nontrivial factor of n . Either a divisor (perhaps composite) or `None` is returned.

The value of a is the base that is used in the test $\gcd(a^{**M} - 1, n)$. The default is 2. If `retries` > 0 then if no factor is found after the first attempt, a new a will be generated randomly (using the seed) and the process repeated.

Note: the value of M is $\text{lcm}(1..B) = \text{reduce}(\text{ilcm}, \text{range}(2, B + 1))$.

A search is made for factors next to even numbers having a power smoothness less than B . Choosing a larger B increases the likelihood of finding a larger factor but takes longer. Whether a factor of n is found or not depends on a and the power smoothness of the even number just less than the factor p (hence the name $p - 1$).

Although some discussion of what constitutes a good a some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if $\gcd(a^{**M} - 1, n) = N$ then $a^{**M} \% q^{**r}$ is 1 for every prime power divisor of N . But consider the following:

```
>>> from sympy.ntheory.factor_ import smoothness_p, pollard_pm1
>>> n=257*1009
>>> smoothness_p(n)
(-1, [(257, (1, 2, 256)), (1009, (1, 7, 16))])
```

So we should (and can) find a root with $B=16$:

```
>>> pollard_pm1(n, B=16, a=3)
1009
```

If we attempt to increase B to 256 we find that it doesn't work:

```
>>> pollard_pm1(n, B=256)
>>>
```

But if the value of a is changed we find that only multiples of 257 work, e.g.:

```
>>> pollard_pm1(n, B=256, a=257)
1009
```

Checking different a values shows that all the ones that didn't work had a gcd value not equal to n but equal to one of the factors:

```
>>> from sympy.core.numbers import ilcm, igcd
>>> from sympy import factorint, Pow
>>> M = 1
>>> for i in range(2, 256):
...     M = ilcm(M, i)
...
>>> set([igcd(pow(a, M, n) - 1, n) for a in range(2, 256) if
...     igcd(pow(a, M, n) - 1, n) != n])
{1009}
```

But does $aM \% d$ for every divisor of n give 1?

```
>>> aM = pow(255, M, n)
>>> [(d, aM%Pow(*d.args)) for d in factorint(n, visual=True).args]
[(257**1, 1), (1009**1, 1)]
```

No, only one of them. So perhaps the principle is that a root will be found for a given value of B provided that:

1. the power smoothness of the $p - 1$ value next to the root does not exceed B
2. $a**M \% p \neq 1$ for any of the divisors of n .

By trying more than one a it is possible that one of them will yield a factor.

References

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 236-238
- <http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node81.html>
- <http://www.cs.toronto.edu/~yuvalf/Factorization.pdf>

Examples

With the default smoothness bound, this number can't be cracked:

```
>>> from sympy.ntheory import pollard_pm1, primefactors
>>> pollard_pm1(21477639576571)
```

Increasing the smoothness bound helps:

```
>>> pollard_pm1(21477639576571, B=2000)
4410317
```

Looking at the smoothness of the factors of this number we find:

```
>>> from sympy.utilities import flatten
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B -pow are the same for the $p - 1$ factorizations of the divisors because those factorizations had a very large prime factor:

```
>>> factorint(4410317 - 1)
{2: 2, 617: 1, 1787: 1}
>>> factorint(4869863-1)
{2: 1, 2434931: 1}
```

Note that until B reaches the B-pow value of 1787, the number is not cracked;

```
>>> pollard_pm1(21477639576571, B=1786)
>>> pollard_pm1(21477639576571, B=1787)
4410317
```

The B value has to do with the factors of the number next to the divisor, not the divisors themselves. A worst case scenario is that the number next to the factor p has a large prime divisor or is a perfect power. If these conditions apply then the power-smoothness will be about $p/2$ or p. The more realistic is that there will be a large prime factor next to p requiring a B value on the order of $p/2$. Although primes may have been searched for up to this level, the $p/2$ is a factor of $p - 1$, something that we don't know. The modular.math reference below states that 15% of numbers in the range of 10^{15} to $15^{15} + 10^4$ are 10^6 power smooth so a B of 10^6 will fail 85% of the time in that range. From 10^8 to $10^8 + 10^3$ the percentages are nearly reversed...but in that range the simple trial division is quite fast.

`sympy.ntheory.factor_.factorint(n, limit=None, use_trial=True, use_rho=True, use_pm1=True, verbose=False, visual=None, multiple=False)`

Given a positive integer n, `factorint(n)` returns a dict containing the prime factors of n as keys and their respective multiplicities as values. For example:

```
>>> from sympy.ntheory import factorint
>>> factorint(2000)      # 2000 = (2**4) * (5**3)
{2: 4, 5: 3}
>>> factorint(65537)    # This number is prime
{65537: 1}
```

For input less than 2, factorint behaves as follows:

- `factorint(1)` returns the empty factorization, `{}`
- `factorint(0)` returns `{0:1}`
- `factorint(-n)` adds `-1:1` to the factors and then factors n

Partial Factorization:

If `limit (> 3)` is specified, the search is stopped after performing trial division up to (and including) the limit (or taking a corresponding number of $\rho/p-1$ steps). This is useful if one has a large number and only is interested in finding small factors (if any). Note that setting a limit does not prevent larger factors from being found early; it simply means that the largest factor may be composite. Since checking for perfect power is relatively cheap, it is done regardless of the limit setting.

This number, for example, has two small factors and a huge semi-prime factor that cannot be reduced easily:

```
>>> from sympy.ntheory import isprime
>>> from sympy.core.compatibility import long
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, long(202916782076162456022877024859): 1, 7: 1}
```

(continues on next page)

(continued from previous page)

```
True
>>> isprime(max(f))
False
```

This number has a small factor and a residual perfect power whose base is greater than the limit:

```
>>> factorint(3*101**7, limit=5)
{3: 1, 101: 7}
```

List of Factors:

If `multiple` is set to `True` then a list containing the prime factors including multiplicities is returned.

```
>>> factorint(24, multiple=True)
[2, 2, 2, 3]
```

Visual Factorization:

If `visual` is set to `True`, then it will return a visual factorization of the integer. For example:

```
>>> from sympy import pprint
>>> pprint(factorint(4200, visual=True))
  3  1  2  1
2 *3 *5 *7
```

Note that this is achieved by using the `evaluate=False` flag in `Mul` and `Pow`. If you do other manipulations with an expression where `evaluate=False`, it may evaluate. Therefore, you should use the visual option only for visualization, and use the normal dictionary returned by `visual=False` if you want to perform operations on the factors.

You can easily switch between the two forms by sending them back to `factorint`:

```
>>> from sympy import Mul, Pow
>>> regular = factorint(1764); regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular))
  2  2  2
2 *3 *7
```

```
>>> visual = factorint(1764, visual=True); pprint(visual)
  2  2  2
2 *3 *7
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:

Input	True	False	other
dict	mul	dict	mul
n	mul	dict	dict
mul	mul	dict	dict

See also:

[smoothness](#) (page 306), [smoothness_p](#) (page 307), [divisors](#) (page 315)

Notes

Algorithm:

The function switches between multiple algorithms. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The Pollard rho and p-1 algorithms are used to find large factors ahead of time; they will often find factors of the order of 10 digits within a few seconds:

```
>>> factors = factorint(12345678910111213141516)
>>> for base, exp in sorted(factors.items()):
...     print('%s %s' % (base, exp))
...
2 2
2507191691 1
1231026625769 1
```

Any of these methods can optionally be disabled with the following boolean parameters:

- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard's rho method
- `use_pm1`: Toggle use of Pollard's p-1 method

`factorint` also periodically checks if the remaining part is a prime number or a perfect power, and in those cases stops.

For unevaluated factorial, it uses Legendre's formula(theorem).

If `verbose` is set to `True`, detailed progress is printed.

`sympy.ntheory.factor_.primefactors(n, limit=None, verbose=False)`

Return a sorted list of `n`'s prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization. Unlike `factorint()`, `primefactors()` does not return -1 or 0.

See also:

[divisors](#) (page 315)

Examples

```
>>> from sympy.ntheory import primefactors, factorint, isprime
>>> primefactors(6)
[2, 3]
>>> primefactors(-5)
[5]
```



```
>>> sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]
>>> primefactors(123456)
[2, 3, 643]
```

```
>>> sorted(factorint(10000000001, limit=200).items())
[(101, 1), (99009901, 1)]
>>> isprime(99009901)
False
>>> primefactors(10000000001, limit=300)
[101]
```

`sympy.ntheory.factor_.divisors(n, generator=False)`

Return all divisors of n sorted from 1.. n by default. If `generator` is `True` an unordered generator is returned.

The number of divisors of n can be quite large if there are many prime factors (counting repeated factors). If only the number of factors is desired use `divisor_count(n)`.

See also:

`primefactors` (page 314), `factorint` (page 312), `divisor_count` (page 315)

Examples

```
>>> from sympy import divisors, divisor_count
>>> divisors(24)
[1, 2, 3, 4, 6, 8, 12, 24]
>>> divisor_count(24)
8
```

```
>>> list(divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60, 120]
```

This is a slightly modified version of Tim Peters referenced at: <http://stackoverflow.com/questions/1010381/python-factorization>

`sympy.ntheory.factor_.divisor_count(n, modulus=1)`

Return the number of divisors of n . If `modulus` is not 1 then only those that are divisible by `modulus` are counted.

See also:

`factorint` (page 312), `divisors` (page 315), `totient` (page 317)

References

- <http://www.mayer.dial.pipex.com/maths/formulae.htm>

```
>>> from sympy import divisor_count
>>> divisor_count(6)
4
```

`sympy.ntheory.factor_.divisors(n, generator=False)`

Return all unitary divisors of n sorted from 1.. n by default. If `generator` is `True` an unordered generator is returned.

The number of unitary divisors of n can be quite large if there are many prime factors. If only the number of unitary divisors is desired use `udivisor_count(n)`.

See also:

`primefactors` (page 314), `factorint` (page 312), `divisors` (page 315), `divisor_count` (page 315), `udivisor_count` (page 316)

References

- http://en.wikipedia.org/wiki/Unitary_divisor
- <http://mathworld.wolfram.com/UnitaryDivisor.html>

Examples

```
>>> from sympy.ntheory.factor_ import udivisors, udivisor_count
>>> udivisors(15)
[1, 3, 5, 15]
>>> udivisor_count(15)
4
```

```
>>> sorted(udivisors(120, generator=True))
[1, 3, 5, 8, 15, 24, 40, 120]
```

`sympy.ntheory.factor_.udivisor_count(n)`
Return the number of unitary divisors of n .

See also:

`factorint` (page 312), `divisors` (page 315), `udivisors` (page 315), `divisor_count` (page 315), `totient` (page 317)

References

- <http://mathworld.wolfram.com/UnitaryDivisorFunction.html>

```
>>> from sympy.ntheory.factor_ import udivisor_count
>>> udivisor_count(120)
8
```

`sympy.ntheory.factor_.antidivisors(n, generator=False)`
Return all antidivisors of n sorted from $1..n$ by default.

Antidivisors [R394] (page 1856) of n are numbers that do not divide n by the largest possible margin. If `generator` is `True` an unordered generator is returned.

See also:

`primefactors` (page 314), `factorint` (page 312), `divisors` (page 315), `divisor_count` (page 315), `antidivisor_count` (page 317)

References

[R394] (page 1856)

Examples

```
>>> from sympy.ntheory.factor_ import antidivisors
>>> antidivisors(24)
[7, 16]
```

```
>>> sorted(antidivisors(128, generator=True))
[3, 5, 15, 17, 51, 85]
```

`sympy.ntheory.factor_.antidivisor_count(n)`

Return the number of antidivisors [\[R395\]](#) (page 1856) of n .

See also:

[factorint](#) (page 312), [divisors](#) (page 315), [antidivisors](#) (page 316), [divisor_count](#) (page 315), [totient](#) (page 317)

References

[\[R395\]](#) (page 1856)

Examples

```
>>> from sympy.ntheory.factor_ import antidivisor_count
>>> antidivisor_count(13)
4
>>> antidivisor_count(27)
5
```

class `sympy.ntheory.factor_.totient`

Calculate the Euler totient function $\phi(n)$

`totient(n)` or $\phi(n)$ is the number of positive integers $\leq n$ that are relatively prime to n .

See also:

[divisor_count](#) (page 315)

References

[\[R396\]](#) (page 1856), [\[R397\]](#) (page 1856)

Examples

```
>>> from sympy.ntheory import totient
>>> totient(1)
1
>>> totient(25)
20
```

class `sympy.ntheory.factor_.reduced_totient`

Calculate the Carmichael reduced totient function $\lambda(n)$

`reduced_totient(n)` or $\lambda(n)$ is the smallest $m > 0$ such that $k^m \equiv 1 \pmod n$ for all k relatively prime to n .

See also:

`totient` (page 317)

References

[R398] (page 1856), [R399] (page 1856)

Examples

```
>>> from sympy.ntheory import reduced_totient
>>> reduced_totient(1)
1
>>> reduced_totient(8)
2
>>> reduced_totient(30)
4
```

class `sympy.ntheory.factor_.divisor_sigma`

Calculate the divisor function $\sigma_k(n)$ for positive integer n

`divisor_sigma(n, k)` is equal to `sum([x**k for x in divisors(n)])`

If n 's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k(n) = \prod_{i=1}^{\omega} (1 + p_i^k + p_i^{2k} + \cdots + p_i^{m_i k}).$$

Parameters **k**: power of divisors in the sum

for $k = 0, 1$: `divisor_sigma(n, 0)` is equal to `divisor_count(n)`

`divisor_sigma(n, 1)` is equal to `sum(divisors(n))`

Default for k is 1.

See also:

`divisor_count` (page 315), `totient` (page 317), `divisors` (page 315), `factorint` (page 312)

References

[R400] (page 1856)

Examples

```
>>> from sympy.ntheory import divisor_sigma
>>> divisor_sigma(18, 0)
6
>>> divisor_sigma(39, 1)
56
>>> divisor_sigma(12, 2)
210
>>> divisor_sigma(37)
38
```

class sympy.ntheory.factor_.**udivisor_sigma**

Calculate the unitary divisor function $\sigma_k^*(n)$ for positive integer n

`udivisor_sigma(n, k)` is equal to `sum([x**k for x in udivisors(n)])`

If n 's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k^*(n) = \prod_{i=1}^{\omega} (1 + p_i^{m_i k}).$$

Parameters **k** : power of divisors in the sum

for $k = 0, 1$: `udivisor_sigma(n, 0)` is equal to `udivisor_count(n)`

`udivisor_sigma(n, 1)` is equal to `sum(udivisors(n))`

Default for k is 1.

See also:

`divisor_count` (page 315), `totient` (page 317), `divisors` (page 315), `udivisors` (page 315), `udivisor_count` (page 316), `divisor_sigma` (page 318), `factorint` (page 312)

References

[R401] (page 1856)

Examples

```
>>> from sympy.ntheory.factor_ import udivisor_sigma
>>> udivisor_sigma(18, 0)
4
>>> udivisor_sigma(74, 1)
114
>>> udivisor_sigma(36, 3)
47450
>>> udivisor_sigma(111)
152
```

`sympy.ntheory.factor_.core(n, t=2)`

Calculate $\text{core}(n, t) = \text{core}_t(n)$ of a positive integer n

$\text{core}_2(n)$ is equal to the squarefree part of n

If n 's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\text{core}_t(n) = \prod_{i=1}^{\omega} p_i^{m_i \bmod t}.$$

Parameters t : $\text{core}(n, t)$ calculates the t -th power free part of n

$\text{core}(n, 2)$ is the squarefree part of n $\text{core}(n, 3)$ is the cubefree part of n

Default for t is 2.

See also:

[factorint](#) (page 312), [sympy.solvers.diophantine.square_factor](#) (page 1328)

References

[R402] (page 1856)

Examples

```
>>> from sympy.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
379238
>>> core(15**11, 10)
15
```

`sympy.ntheory.factor_.digits(n, b=10)`

Return a list of the digits of n in base b . The first element in the list is b (or $-b$ if n is negative).

Examples

```
>>> from sympy.ntheory.factor_ import digits
>>> digits(35)
[10, 3, 5]
>>> digits(27, 2)
[2, 1, 1, 0, 1, 1]
>>> digits(65536, 256)
```

(continues on next page)

(continued from previous page)

```
[256, 1, 0, 0]
>>> digits(-3958, 27)
[-27, 5, 11, 16]
```

class sympy.ntheory.factor_.primenu

Calculate the number of distinct prime factors for a positive integer n .

If n 's prime factorization is:

$$n = \prod_{i=1}^k p_i^{m_i},$$

then `primenu(n)` or $\nu(n)$ is:

$$\nu(n) = k.$$

See also:

`factorint` (page 312)

References

[R403] (page 1856)

Examples

```
>>> from sympy.ntheory.factor_ import primenu
>>> primenu(1)
0
>>> primenu(30)
3
```

class sympy.ntheory.factor_.primeomega

Calculate the number of prime factors counting multiplicities for a positive integer n .

If n 's prime factorization is:

$$n = \prod_{i=1}^k p_i^{m_i},$$

then `primeomega(n)` or $\Omega(n)$ is:

$$\Omega(n) = \sum_{i=1}^k m_i.$$

See also:

`factorint` (page 312)

References

[R404] (page 1856)

Examples

```
>>> from sympy.ntheory.factor_ import primeomega
>>> primeomega(1)
0
>>> primeomega(20)
3
```

`sympy.ntheory.modular.symmetric_residue(a, m)`

Return the residual mod m such that it is within half of the modulus.

```
>>> from sympy.ntheory.modular import symmetric_residue
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

`sympy.ntheory.modular.crt(m, v, symmetric=False, check=True)`

Chinese Remainder Theorem.

The moduli in m are assumed to be pairwise coprime. The output is then an integer f , such that $f = v_i \bmod m_i$ for each pair out of v and m . If `symmetric` is `False` a positive integer will be returned, else $|f|$ will be less than or equal to the LCM of the moduli, and thus f may be negative.

If the moduli are not co-prime the correct result will be returned if/when the test of the result is found to be incorrect. This result will be `None` if there is no solution.

The keyword `check` can be set to `False` if it is known that the moduli are coprime.

As an example consider a set of residues $U = [49, 76, 65]$ and a set of moduli $M = [99, 97, 95]$. Then we have:

```
>>> from sympy.ntheory.modular import crt, solve_congruence
>>> crt([99, 97, 95], [49, 76, 65])
(639985, 912285)
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

If the moduli are not co-prime, you may receive an incorrect result if you use `check=False`:

```
>>> crt([12, 6, 17], [3, 4, 2], check=False)
(954, 1224)
>>> [954 % m for m in [12, 6, 17]]
[6, 0, 2]
>>> crt([12, 6, 17], [3, 4, 2]) is None
True
>>> crt([3, 6], [2, 5])
(5, 6)
```

Note: the order of `gf_crt`'s arguments is reversed relative to `crt`, and that `solve_congruence` takes residue, modulus pairs.

Programmer's note: rather than checking that all pairs of moduli share no GCD (an $O(n^2)$ test) and rather than factoring all moduli and seeing that there is no factor in

common, a check that the result gives the indicated residuals is performed - an $O(n)$ operation.

See also:

`solve_congruence` (page 323)

`sympy.polys.galoistools.gf_crt` (page 954) low level crt routine used by this routine

`sympy.ntheory.modular.crt1(m)`

First part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from sympy.ntheory.modular import crt1
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`sympy.ntheory.modular.crt2(m, v, mm, e, s, symmetric=False)`

Second part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from sympy.ntheory.modular import crt1, crt2
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

`sympy.ntheory.modular.solve_congruence(*remainder_modulus_pairs, **hint)`

Compute the integer n that has the residual a_i when it is divided by m_i where the a_i and m_i are given as pairs to this function: $((a_1, m_1), (a_2, m_2), \dots)$. If there is no solution, return None. Otherwise return n and its modulus.

The m_i values need not be co-prime. If it is known that the moduli are not co-prime then the hint check can be set to False (default=True) and the check for a quicker solution via `crt()` (valid when the moduli are co-prime) will be skipped.

If the hint `symmetric` is True (default is False), the value of n will be within $1/2$ of the modulus, possibly negative.

See also:

`crt` (page 322) high level routine implementing the Chinese Remainder Theorem

Examples

```
>>> from sympy.ntheory.modular import solve_congruence
```

What number is 2 mod 3, 3 mod 5 and 2 mod 7?

```
>>> solve_congruence((2, 3), (3, 5), (2, 7))
(23, 105)
>>> [23 % m for m in [3, 5, 7]]
[2, 3, 2]
```

If you prefer to work with all remainder in one list and all moduli in another, send the arguments like this:

```
>>> solve_congruence(*zip((2, 3, 2), (3, 5, 7)))
(23, 105)
```

The moduli need not be co-prime; in this case there may or may not be a solution:

```
>>> solve_congruence((2, 3), (4, 6)) is None
True
```

```
>>> solve_congruence((2, 3), (5, 6))
(5, 6)
```

The symmetric flag will make the result be within 1/2 of the modulus:

```
>>> solve_congruence((2, 3), (5, 6), symmetric=True)
(-1, 6)
```

`sympy.ntheory.multinomial.binomial_coefficients(n)`

Return a dictionary containing pairs $(k_1, k_2) : C_{k_1, k_2} n$ where $C_{k_1, k_2} n$ are binomial coefficients and $n = k_1 + k_2$. Examples =====

```
>>> from sympy.ntheory import binomial_coefficients
>>> binomial_coefficients(9)
{(0, 9): 1, (1, 8): 9, (2, 7): 36, (3, 6): 84,
 (4, 5): 126, (5, 4): 126, (6, 3): 84, (7, 2): 36, (8, 1): 9, (9, 0): 1}
```

See also:

[binomial_coefficients_list](#) (page 324), [multinomial_coefficients](#) (page 324)

`sympy.ntheory.multinomial.binomial_coefficients_list(n)`

Return a list of binomial coefficients as rows of the Pascal's triangle.

See also:

[binomial_coefficients](#) (page 324), [multinomial_coefficients](#) (page 324)

Examples

```
>>> from sympy.ntheory import binomial_coefficients_list
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

`sympy.ntheory.multinomial.multinomial_coefficients(m, n)`

Return a dictionary containing pairs $\{(k_1, k_2, \dots, k_m) : C_{k_1, k_2, \dots, k_m} n\}$ where $C_{k_1, k_2, \dots, k_m} n$ are multinomial coefficients such that $n = k_1 + k_2 + \dots + k_m$.

For example:

```
>>> from sympy.ntheory import multinomial_coefficients
>>> multinomial_coefficients(2, 5) # indirect doctest
{(0, 5): 1, (1, 4): 5, (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
```

The algorithm is based on the following result:

$$\binom{n}{k_1, \dots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^m \binom{n}{k_1 + 1, \dots, k_i - 1, \dots}$$

Code contributed to Sage by Yann Laigle-Chapuy, copied with permission of the author.

See also:

[binomial_coefficients_list](#) (page 324), [binomial_coefficients](#) (page 324)

`sympy.ntheory.multinomial.multinomial_coefficients_iterator(m, n, _tuple=<class 'tuple'>)`

multinomial coefficient iterator

This routine has been optimized for m large with respect to n by taking advantage of the fact that when the monomial tuples t are stripped of zeros, their coefficient is the same as that of the monomial tuples from `multinomial_coefficients(n, n)`. Therefore, the latter coefficients are precomputed to save memory and time.

```
>>> from sympy.ntheory.multinomial import multinomial_coefficients
>>> m53, m33 = multinomial_coefficients(5,3), multinomial_coefficients(3,3)
>>> m53[(0,0,0,1,2)] == m53[(0,0,1,0,2)] == m53[(1,0,2,0,0)] == m33[(0,1,2)]
True
```

Examples

```
>>> from sympy.ntheory.multinomial import multinomial_coefficients_iterator
>>> it = multinomial_coefficients_iterator(20,3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1)
```

`sympy.ntheory.partitions_.npartitions(n, verbose=False)`

Calculate the partition function $P(n)$, i.e. the number of ways that n can be written as a sum of positive integers.

$P(n)$ is computed using the Hardy-Ramanujan-Rademacher formula [\[R405\]](#) (page 1856).

The correctness of this implementation has been tested through $10^{*}10$.

References

[\[R405\]](#) (page 1856)

Examples

```
>>> from sympy.ntheory import npartitions
>>> npartitions(25)
1958
```

`sympy.ntheory.primetest.mr(n, bases)`

Perform a Miller-Rabin strong pseudoprime test on n using a given list of bases/witnesses.

References

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 135-138

A list of thresholds and the bases they require are here: http://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants_of_the_test

Examples

```
>>> from sympy.ntheory.primetest import mr
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

`sympy.ntheory.primetest.isprime(n)`

Test if n is a prime number (True) or not (False). For $n < 2^{64}$ the answer is definitive; larger n values have a small probability of actually being pseudoprimes.

Negative numbers (e.g. -2) are not considered prime.

The first step is looking for trivial factors, which if found enables a quick return. Next, if the sieve is large enough, use bisection search on the sieve. For small numbers, a set of deterministic Miller-Rabin tests are performed with bases that are known to have no counterexamples in their range. Finally if the number is larger than 2^{64} , a strong BPSW test is performed. While this is a probable prime test and we believe counterexamples exist, there are no known counterexamples.

See also:

`sympy.ntheory.generate.primerange` (page 302) Generates all primes in a given range

`sympy.ntheory.generate.primepi` (page 301) Return the number of primes less than or equal to n

`sympy.ntheory.generate.prime` (page 300) Return the n th prime

References

- http://en.wikipedia.org/wiki/Strong_pseudoprime
- “Lucas Pseudoprimes”, Baillie and Wagstaff, 1980. <http://mpqs.free.fr/LucasPseudoprimes.pdf>
- https://en.wikipedia.org/wiki/Baillie-PSW_primality_test

Examples

```
>>> from sympy.ntheory import isprime
>>> isprime(13)
True
>>> isprime(15)
False
```

`sympy.ntheory.residue_ntheory.n_order(a, n)`

Returns the order of a modulo n .

The order of a modulo n is the smallest integer k such that a^k leaves a remainder of 1 with n .

Examples

```
>>> from sympy.ntheory import n_order
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

`sympy.ntheory.residue_ntheory.is_primitive_root(a, p)`

Returns True if a is a primitive root of p

a is said to be the primitive root of p if $\gcd(a, p) == 1$ and $\text{totient}(p)$ is the smallest positive number $s.t.$

$$a^{\text{totient}(p)} \equiv 1 \pmod{p}$$

Examples

```
>>> from sympy.ntheory import is_primitive_root, n_order, totient
>>> is_primitive_root(3, 10)
True
>>> is_primitive_root(9, 10)
False
>>> n_order(3, 10) == totient(10)
True
>>> n_order(9, 10) == totient(10)
False
```

`sympy.ntheory.residue_ntheory.primitive_root(p)`

Returns the smallest primitive root or None

Parameters p : positive integer

References

[R406] (page 1856), [R407] (page 1856)

Examples

```
>>> from sympy.ntheory.residue_ntheory import primitive_root
>>> primitive_root(19)
2
```

`sympy.ntheory.residue_ntheory.sqrt_mod(a, p, all_roots=False)`

Find a root of $x^2 = a \pmod{p}$

Parameters a : integer

p : positive integer

all_roots : if True the list of roots is returned or None

Notes

If there is no root it is returned None; else the returned root is less or equal to $p // 2$; in general is not the smallest one. It is returned $p // 2$ only if it is the only root.

Use `all_roots` only when it is expected that all the roots fit in memory; otherwise use `sqrt_mod_iter`.

Examples

```
>>> from sympy.ntheory import sqrt_mod
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

`sympy.ntheory.residue_ntheory.quadratic_residues(p)`
Returns the list of quadratic residues.

Examples

```
>>> from sympy.ntheory.residue_ntheory import quadratic_residues
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

`sympy.ntheory.residue_ntheory.nthroot_mod(a, n, p, all_roots=False)`
Find the solutions to $x^n = a \pmod p$

Parameters **a** : integer

n : positive integer

p : positive integer

all_roots : if False returns the smallest root, else the list of roots

Examples

```
>>> from sympy.ntheory.residue_ntheory import nthroot_mod
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

`sympy.ntheory.residue_ntheory.is_nthpow_residue(a, n, m)`
Returns True if $x^n = a \pmod m$ has solutions.

References

[R408] (page 1856)

`sympy.ntheory.residue_ntheory.is_quad_residue(a, p)`

Returns True if $a \pmod{p}$ is in the set of squares mod p , i.e. $a \% p$ in $\text{set}([i**2 \% p \text{ for } i \text{ in range}(p)])$. If p is an odd prime, an iterative method is used to make the determination:

```
>>> from sympy.ntheory import is_quad_residue
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
>>> [j for j in range(7) if is_quad_residue(j, 7)]
[0, 1, 2, 4]
```

See also:

[legendre_symbol](#) (page 329), [jacobi_symbol](#) (page 329)

`sympy.ntheory.residue_ntheory.legendre_symbol(a, p)`

Returns the Legendre symbol (a/p) .

For an integer a and an odd prime p , the Legendre symbol is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \text{ divides } a \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \end{cases}$$

Parameters a : integer

p : odd prime

See also:

[is_quad_residue](#) (page 328), [jacobi_symbol](#) (page 329)

Examples

```
>>> from sympy.ntheory import legendre_symbol
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, 1, -1, -1]
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
```

`sympy.ntheory.residue_ntheory.jacobi_symbol(m, n)`

Returns the Jacobi symbol (m/n) .

For any integer m and any positive odd integer n the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n :

$$\left(\frac{m}{n}\right) = \left(\frac{m}{p^1}\right)^{\alpha_1} \left(\frac{m}{p^2}\right)^{\alpha_2} \cdots \left(\frac{m}{p^k}\right)^{\alpha_k} \text{ where } n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

Like the Legendre symbol, if the Jacobi symbol $(\frac{m}{n}) = -1$ then m is a quadratic nonresidue modulo n .

But, unlike the Legendre symbol, if the Jacobi symbol $(\frac{m}{n}) = 1$ then m may or may not be a quadratic residue modulo n .

Parameters m : integer

n : odd positive integer

See also:

[is_quad_residue](#) (page 328), [legendre_symbol](#) (page 329)

Examples

```
>>> from sympy.ntheory import jacobi_symbol, legendre_symbol
>>> from sympy import Mul, S
>>> jacobi_symbol(45, 77)
-1
>>> jacobi_symbol(60, 121)
1
```

The relationship between the `jacobi_symbol` and `legendre_symbol` can be demonstrated as follows:

```
>>> L = legendre_symbol
>>> S(45).factors()
{3: 2, 5: 1}
>>> jacobi_symbol(7, 45) == L(7, 3)**2 * L(7, 5)**1
True
```

`sympy.ntheory.residue_ntheory.discrete_log(n, a, b, order=None, prime_order=None)`

Compute the discrete logarithm of a to the base b modulo n .

This is a recursive function to reduce the discrete logarithm problem in cyclic groups of composite order to the problem in cyclic groups of prime order.

It employs different algorithms depending on the problem (subgroup order size, prime order or not):

- Trial multiplication
- Baby-step giant-step
- Pollard's Rho
- Pohlig-Hellman

References

[R409] (page 1856), [R410] (page 1856)

Examples

```
>>> from sympy.ntheory import discrete_log
>>> discrete_log(41, 15, 7)
3
```

`sympy.ntheory.continued_fraction.continued_fraction_convergents(cf)`

Return an iterator over the convergents of a continued fraction (`cf`).

The parameter should be an iterable returning successive partial quotients of the continued fraction, such as might be returned by `continued_fraction_iterator`. In computing the convergents, the continued fraction need not be strictly in canonical form (all integers, all but the first positive). Rational and negative elements may be present in the expansion.

See also:

[continued_fraction_iterator](#) (page 331)

Examples

```
>>> from sympy.core import Rational, pi
>>> from sympy import S
>>> from sympy.ntheory.continued_fraction import continued_fraction_
    ↪ convergents, continued_fraction_iterator
```

```
>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]
```

```
>>> list(continued_fraction_convergents([1, S('1/2'), -7, S('1/4')]))
[1, 3, 19/5, 7]
```

```
>>> it = continued_fraction_convergents(continued_fraction_iterator(pi))
>>> for n in range(7):
...     print(next(it))
3
22/7
333/106
355/113
103993/33102
104348/33215
208341/66317
```

`sympy.ntheory.continued_fraction.continued_fraction_iterator(x)`
Return continued fraction expansion of `x` as iterator.

References

[R411] (page 1856)

Examples

```
>>> from sympy.core import Rational, pi
>>> from sympy.ntheory.continued_fraction import continued_fraction_iterator
```

```
>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]
>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 1, 2]
```

```
>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
292
1
```

(continues on next page)

(continued from previous page)

```
1
1
```

`sympy.ntheory.continued_fraction.continued_fraction_periodic(p, q, d=0)`

Find the periodic continued fraction expansion of a quadratic irrational.

Compute the continued fraction expansion of a rational or a quadratic irrational number, i.e. $\frac{p+\sqrt{d}}{q}$, where p , q and $d \geq 0$ are integers.

Returns the continued fraction representation (canonical form) as a list of integers, optionally ending (for quadratic irrationals) with repeating block as the last term of this list.

Parameters **p** : int

the rational part of the number's numerator

q : int

the denominator of the number

d : int, optional

the irrational part (discriminator) of the number's numerator

See also:

[continued_fraction_iterator](#) (page 331), [continued_fraction_reduce](#) (page 332)

References

[R412] (page 1856), [R413] (page 1856)

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, 1]]
```

Golden ratio has the simplest continued fraction expansion:

```
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

`sympy.ntheory.continued_fraction.continued_fraction_reduce(cf)`

Reduce a continued fraction to a rational or quadratic irrational.

Compute the rational or quadratic irrational number from its terminating or periodic continued fraction expansion. The continued fraction expansion (cf) should be supplied as a terminating iterator supplying the terms of the expansion. For terminating continued fractions, this is equivalent to `list(continued_fraction_convergents(cf))[-1]`,

only a little more efficient. If the expansion has a repeating part, a list of the repeating terms should be returned as the last element from the iterator. This is the format returned by `continued_fraction_periodic`.

For quadratic irrationals, returns the largest solution found, which is generally the one sought, if the fraction is in canonical form (all terms positive except possibly the first).

See also:

`continued_fraction_periodic` (page 332)

Examples

```
>>> from sympy.ntheory.continued_fraction import continued_fraction_reduce
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]).n(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
1/2 + sqrt(5)/2
>>> from sympy.ntheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```

class `sympy.ntheory.mobius`

Möbius function maps natural number to $\{-1, 0, 1\}$

It is defined as follows:

1. 1 if $n = 1$.
2. 0 if n has a squared prime factor.
3. $(-1)^k$ if n is a square-free positive integer with k number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Möbius Function model).

Parameters `n` : positive integer

References

[R414] (page 1856), [R415] (page 1856)

Examples

```
>>> from sympy.ntheory import mobius
>>> mobius(13*7)
1
>>> mobius(1)
1
```

(continues on next page)

(continued from previous page)

```
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```

`sympy.ntheory.egyptian_fraction.egyptian_fraction(r, algorithm='Greedy')`
 Return the list of denominators of an Egyptian fraction expansion [R416] (page 1856) of the said rational r .

Parameters r : Rational

a positive rational number.

algorithm : { "Greedy", "Graham Jewett", "Takenouchi", "Golomb" }, optional

Denotes the algorithm to be used (the default is "Greedy").

See also:

`sympy.core.numbers.Rational` (page 141)

Notes

Currently the following algorithms are supported:

1. Greedy Algorithm

Also called the Fibonacci-Sylvester algorithm [R417] (page 1856). At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

It has some distinct properties:

- (a) Given p/q in lowest terms, generates an expansion of maximum length p . Even as the numerators get large, the number of terms is seldom more than a handful.
- (b) Uses minimal memory.
- (c) The terms can blow up (standard examples of this are $5/121$ and $31/311$). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

2. Graham Jewett Algorithm

The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always $2^{*(x/\gcd(x, y)) - 1}$. See [R418] (page 1856).

3. Takenouchi Algorithm

The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates. See [R418] (page 1856).

4. Golomb's Algorithm

A method given by Golomb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972). See [R419] (page 1856).

If the given rational is greater than or equal to 1, a greedy algorithm of summing the harmonic sequence $1/1 + 1/2 + 1/3 + \dots$ is used, taking all the unit fractions of this sequence until adding one more would be greater than the given number. This list of

denominators is prefixed to the result from the requested algorithm used on the remainder. For example, if r is $8/3$, using the Greedy algorithm, we get $[1, 2, 3, 4, 5, 6, 7, 14, 420]$, where the beginning of the sequence, $[1, 2, 3, 4, 5, 6, 7]$ is part of the harmonic sequence summing to $363/140$, leaving a remainder of $31/420$, which yields $[14, 420]$ by the Greedy algorithm. The result of `egyptian_fraction(Rational(8, 3), "Golomb")` is $[1, 2, 3, 4, 5, 6, 7, 14, 574, 2788, 6460, 11590, 33062, 113820]$, and so on.

References

[R416] (page 1856), [R417] (page 1856), [R418] (page 1856), [R419] (page 1856)

Examples

```
>>> from sympy import Rational
>>> from sympy.ntheory.egyptian_fraction import egyptian_fraction
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction(Rational(3, 7), "Graham Jewett")
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction(Rational(3, 7), "Takenouchi")
[4, 7, 28]
>>> egyptian_fraction(Rational(3, 7), "Golomb")
[3, 15, 35]
>>> egyptian_fraction(Rational(11, 5), "Golomb")
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

5.4 Basic Cryptography Module

Warning: This module is intended for educational purposes only. Do not use the functions in this module for real cryptographic applications. If you wish to encrypt real data, we recommend using something like the `cryptography` module.

Encryption is the process of hiding a message and a cipher is a means of doing so. Included in this module are both block and stream ciphers:

- Shift cipher
- Affine cipher
- substitution ciphers
- Vigenere's cipher
- Hill's cipher
- Bifid ciphers
- RSA
- Kid RSA
- linear-feedback shift registers (for stream ciphers)
- ElGamal encryption

In a substitution cipher “units” (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

A transposition cipher is a method of encryption by which the positions held by “units” of plaintext are replaced by a permutation of the plaintext. That is, the order of the units is changed using a bijective function on the position of the characters to perform the encryption.

A monoalphabetic cipher uses fixed substitution over the entire message, whereas a polyalphabetic cipher uses a number of substitutions at different times in the message.

`sympy.crypto.crypto.AZ(s=None)`

Return the letters of `s` in uppercase. In case more than one string is passed, each of them will be processed and a list of upper case strings will be returned.

See also:

`check_and_join` (page 336)

Examples

```
>>> from sympy.crypto.crypto import AZ
>>> AZ('Hello, world!')
'HELLOWORLD'
>>> AZ('Hello, world!'.split())
['HELLO', 'WORLD']
```

`sympy.crypto.crypto.padded_key(key, symbols, filter=True)`

Return a string of the distinct characters of `symbols` with those of `key` appearing first, omitting characters in `key` that are not in `symbols`. A `ValueError` is raised if a) there are duplicate characters in `symbols` or b) there are characters in `key` that are not in `symbols`.

Examples

```
>>> from sympy.crypto.crypto import padded_key
>>> padded_key('PUPPY', 'OPQRSTUVWXYZ')
'PUYQQRSTVWX'
>>> padded_key('RSA', 'ARTIST')
Traceback (most recent call last):
...
ValueError: duplicate characters in symbols: T
```

`sympy.crypto.crypto.check_and_join(phrase, symbols=None, filter=None)`

Joins characters of *phrase* and if `symbols` is given, raises an error if any character in `phrase` is not in `symbols`.

Parameters `phrase`: string or list of strings to be returned as a string

symbols: iterable of characters allowed in “phrase”;

if `symbols` is `None`, no checking is performed

Examples

```
>>> from sympy.crypto.crypto import check_and_join
>>> check_and_join('a phrase')
'a phrase'
>>> check_and_join('a phrase'.upper().split())
'APHRASE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE', filter=True)
'ARAE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE')
Traceback (most recent call last):
...
ValueError: characters in phrase but not symbols: "!HPS"
```

`sympy.crypto.crypto.cycle_list(k, n)`

Returns the elements of the list `range(n)` shifted to the left by `k` (so the list starts with `k (mod n)`).

Examples

```
>>> from sympy.crypto.crypto import cycle_list
>>> cycle_list(3, 10)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

`sympy.crypto.crypto.encipher_shift(msg, key, symbols=None)`

Performs shift cipher encryption on plaintext `msg`, and returns the ciphertext.

Notes

The shift cipher is also called the Caesar cipher, after Julius Caesar, who, according to Suetonius, used it with a shift of three to protect messages of military significance. Caesar's nephew Augustus reportedly used a similar cipher, but with a right shift of 1.

ALGORITHM:

INPUT:

key: an integer (the secret key)

msg: plaintext of upper-case letters

OUTPUT:

ct: ciphertext of upper-case letters

STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string `msg` a list `L1` of corresponding integers.
2. Compute from the list `L1` a new list `L2`, given by adding $(k \bmod 26)$ to each element in `L1`.
3. Compute from the list `L2` a string `ct` of corresponding letters.

Examples

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

There is also a convenience function that does this with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.decipher_shift(msg, key, symbols=None)`

Return the text by shifting the characters of `msg` to the left by the amount given by `key`.

Examples

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

Or use this function with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.encipher_affine(msg, key, symbols=None, _inverse=False)`

Performs the affine cipher encryption on plaintext `msg`, and returns the ciphertext.

Encryption is based on the map $x \rightarrow ax + b \pmod{N}$ where N is the number of characters in the alphabet. Decryption is based on the map $x \rightarrow cx + d \pmod{N}$, where $c = a^{-1} \pmod{N}$ and $d = -a^{-1}b \pmod{N}$. In particular, for the map to be invertible, we need $\gcd(a, N) = 1$ and an error will be raised if this is not true.

See also:

[`decipher\_affine`](#) (page 339)

Notes

This is a straightforward generalization of the shift cipher with the added complexity of requiring 2 characters to be deciphered in order to recover the key.

ALGORITHM:

INPUT:

msg: string of characters that appear in symbols

a, b: a pair integers, with $\gcd(a, N) = 1$ (the secret key)

symbols: string of characters (default = uppercase letters). When no symbols are given, msg is converted to upper case letters and all other charactes are ignored.

OUTPUT:

ct: string of characters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string msg a list L1 of corresponding integers.
2. Compute from the list L1 a new list L2, given by replacing x by $a*x + b \pmod{N}$, for each element x in L1.
3. Compute from the list L2 a string ct of corresponding letters.

`sympy.crypto.crypto.decipher_affine(msg, key, symbols=None)`

Return the deciphered text that was made from the mapping, $x \rightarrow ax + b \pmod{N}$, where N is the number of characters in the alphabet. Deciphering is done by reciphering with a new key: $x \rightarrow cx + d \pmod{N}$, where $c = a^{-1} \pmod{N}$ and $d = -a^{-1}b \pmod{N}$.

Examples

```
>>> from sympy.crypto.crypto import encipher_affine, decipher_affine
>>> msg = "GO NAVY BEAT ARMY"
>>> key = (3, 1)
>>> encipher_affine(msg, key)
'TROBMVENBGBALV'
>>> decipher_affine(_, key)
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.encipher_substitution(msg, old, new=None)`

Returns the ciphertext obtained by replacing each character that appears in old with the corresponding character in new. If old is a mapping, then new is ignored and the replacements defined by old are used.

Notes

This is a more general than the affine cipher in that the key can only be recovered by determining the mapping for each symbol. Though in practice, once a few symbols are recognized the mappings for other characters can be quickly guessed.

Examples

```
>>> from sympy.crypto.crypto import encipher_substitution, AZ
>>> old = 'OEYAG'
>>> new = '034^6'
>>> msg = AZ("go navy! beat army!")
>>> ct = encipher_substitution(msg, old, new); ct
'60N^V4B3^T^RM4'
```

To decrypt a substitution, reverse the last two arguments:

```
>>> encipher_substitution(ct, new, old)
'GONAVYBEATARMY'
```

In the special case where old and new are a permutation of order 2 (representing a transposition of characters) their order is immaterial:

```
>>> old = 'NAVY'
>>> new = 'ANYV'
>>> encipher = lambda x: encipher_substitution(x, old, new)
>>> encipher('NAVY')
'ANYV'
>>> encipher(_)
'NAVY'
```

The substitution cipher, in general, is a method whereby “units” (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

```
>>> ords = dict(zip('abc', ['\\%i' % ord(i) for i in 'abc']))
>>> print(encipher_substitution('abc', ords))
\97\98\99
```

`sympy.crypto.crypto.encipher_vigenere(msg, key, symbols=None)`

Performs the Vigenère cipher encryption on plaintext msg, and returns the ciphertext.

Notes

The Vigenère cipher is named after Blaise de Vigenère, a sixteenth century diplomat and cryptographer, by a historical accident. Vigenère actually invented a different and more complicated cipher. The so-called Vigenère cipher was actually invented by Giovan Batista Belaso in 1553.

This cipher was used in the 1800’s, for example, during the American Civil War. The Confederacy used a brass cipher disk to implement the Vigenère cipher (now on display in the NSA Museum in Fort Meade) [\[R77\]](#) (page 1856).

The Vigenère cipher is a generalization of the shift cipher. Whereas the shift cipher shifts each letter by the same amount (that amount being the key of the shift cipher) the Vigenère cipher shifts a letter by an amount determined by the key (which is a word or phrase known only to the sender and receiver).

For example, if the key was a single letter, such as “C”, then the so-called Vigenere cipher is actually a shift cipher with a shift of 2 (since “C” is the 2nd letter of the alphabet, if you start counting at 0). If the key was a word with two letters, such as “CA”, then the so-called Vigenère cipher will shift letters in even positions by 2 and letters in odd positions are left alone (shifted by 0, since “A” is the 0th letter, if you start counting at 0).

ALGORITHM:

INPUT:

msg: string of characters that appear in symbols (the plaintext)
key: a string of characters that appear in symbols (the secret key)
symbols: a string of letters defining the alphabet

OUTPUT:

ct: string of characters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string key a list L1 of corresponding integers. Let $n1 = \text{len}(L1)$.
2. Compute from the string msg a list L2 of corresponding integers. Let $n2 = \text{len}(L2)$.
3. Break L2 up sequentially into sublists of size n1; the last sublist may be smaller than n1
4. For each of these sublists L of L2, compute a new list C given by $C[i] = L[i] + L1[i] \pmod{N}$ to the i-th element in the sublist, for each i.
5. Assemble these lists C by concatenation into a new list of length n2.
6. Compute from the new list a string ct of corresponding letters.

Once it is known that the key is, say, n characters long, frequency analysis can be applied to every n -th letter of the ciphertext to determine the plaintext. This method is called Kasiski examination (although it was first discovered by Babbage). If the key is as long as the message and is comprised of randomly selected characters – a one-time pad – the message is theoretically unbreakable.

The cipher Vigenère actually discovered is an “auto-key” cipher described as follows.

ALGORITHM:

INPUT:

key: a string of letters (the secret key)
msg: string of letters (the plaintext message)

OUTPUT:

ct: string of upper-case letters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string msg a list L2 of corresponding integers. Let $n2 = \text{len}(L2)$.
2. Let n1 be the length of the key. Append to the string key the first $n2 - n1$ characters of the plaintext message. Compute from this string (also of length n2) a list L1 of integers corresponding to the letter numbers in the first step.
3. Compute a new list C given by $C[i] = L1[i] + L2[i] \pmod{N}$.
4. Compute from the new list a string ct of letters corresponding to the new integers.

To decipher the auto-key ciphertext, the key is used to decipher the first `n1` characters and then those characters become the key to decipher the next `n1` characters, etc....:

```
>>> m = AZ('go navy, beat army! yes you can'); m
'GONAVYBEATARMYYESYOU CAN'
>>> key = AZ('gold bug'); n1 = len(key); n2 = len(m)
>>> auto_key = key + m[:n2 - n1]; auto_key
'GOLDBUGGONAVYBEATARMYYE'
>>> ct = encipher_vigenere(m, auto_key); ct
'MCYDWSHKGAMKZCELYFGAYR'
>>> n1 = len(key)
>>> pt = []
>>> while ct:
...     part, ct = ct[:n1], ct[n1:]
...     pt.append(decipher_vigenere(part, key))
...     key = pt[-1]
...
>>> ''.join(pt) == m
True
```

References

[R77] (page 1856), [R78] (page 1856)

Examples

```
>>> from sympy.crypto.crypto import encipher_vigenere, AZ
>>> key = "encrypt"
>>> msg = "meet me on monday"
>>> encipher_vigenere(msg, key)
'QRGKKT HRZQE BPR'
```

Section 1 of the Kryptos sculpture at the CIA headquarters uses this cipher and also changes the order of the the alphabet [R78] (page 1856). Here is the first line of that section of the sculpture:

```
>>> from sympy.crypto.crypto import decipher_vigenere, padded_key
>>> alp = padded_key('KRYPTOS', AZ())
>>> key = 'PALIMPSEST'
>>> msg = 'EMUFPHZLR FAXYUSDJKZLDKRNSHGNFIVJ'
>>> decipher_vigenere(msg, key, alp)
'BETWEENSUBTLESHADINGANDTHEABSENC'
```

`sympy.crypto.crypto.decipher_vigenere(msg, key, symbols=None)`
Decode using the Vigenère cipher.

Examples

```
>>> from sympy.crypto.crypto import decipher_vigenere
>>> key = "encrypt"
>>> ct = "QRGK kt HRZQE BPR"
>>> decipher_vigenere(ct, key)
'MEETMEONMONDAY'
```

`sympy.crypto.crypto.encipher_hill(msg, key, symbols=None, pad='Q')`
 Return the Hill cipher encryption of msg.

See also:

`decipher_hill` (page 343)

Notes

The Hill cipher [R79] (page 1856), invented by Lester S. Hill in the 1920's [R80] (page 1856), was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once. The following discussion assumes an elementary knowledge of matrices.

First, each letter is first encoded as a number starting with 0. Suppose your message *msg* consists of n capital letters, with no spaces. This may be regarded an n -tuple M of elements of Z_{26} (if the letters are those of the English alphabet). A key in the Hill cipher is a $k \times k$ matrix K , all of whose entries are in Z_{26} , such that the matrix K is invertible (i.e., the linear transformation $K : Z_N^k \rightarrow Z_N^k$ is one-to-one).

ALGORITHM:

INPUT:

msg: plaintext message of n upper-case letters

key: a $k \times k$ invertible matrix K , all of whose entries are in Z_{26} (or whatever number of symbols are being used).

pad: character (default "Q") to use to make length of text be a multiple of k

OUTPUT:

ct: ciphertext of upper-case letters

STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string msg a list L of corresponding integers. Let $n = \text{len}(L)$.
2. Break the list L up into $t = \text{ceiling}(n/k)$ sublists $L_1, \dots, L_t$ of size k (with the last list "padded" to ensure its size is k).
3. Compute new list $C_1, \dots, C_t$ given by $C[i] = K * L_i$ (arithmetic is done mod N), for each i .
4. Concatenate these into a list $C = C_1 + \dots + C_t$.
5. Compute from C a string ct of corresponding letters. This has length $k*t$.

References

[R79] (page 1856), [R80] (page 1856)

`sympy.crypto.crypto.decipher_hill(msg, key, symbols=None)`
 Deciphering is the same as enciphering but using the inverse of the key matrix.

Examples

```
>>> from sympy.crypto.crypto import encipher_hill, decipher_hill
>>> from sympy import Matrix
```

```
>>> key = Matrix([[1, 2], [3, 5]])
>>> encipher_hill("meet me on monday", key)
'UEQDUEODOCTCWQ'
>>> decipher_hill(_, key)
'MEETMEONMONDAY'
```

When the length of the plaintext (stripped of invalid characters) is not a multiple of the key dimension, extra characters will appear at the end of the enciphered and deciphered text. In order to decipher the text, those characters must be included in the text to be deciphered. In the following, the key has a dimension of 4 but the text is 2 short of being a multiple of 4 so two characters will be added.

```
>>> key = Matrix([[1, 1, 1, 2], [0, 1, 1, 0],
...              [2, 2, 3, 4], [1, 1, 0, 1]])
>>> msg = "ST"
>>> encipher_hill(msg, key)
'HJEB'
>>> decipher_hill(_, key)
'STQQ'
>>> encipher_hill(msg, key, pad="Z")
'ISPK'
>>> decipher_hill(_, key)
'STZZ'
```

If the last two characters of the ciphertext were ignored in either case, the wrong plaintext would be recovered:

```
>>> decipher_hill("HD", key)
'ORMV'
>>> decipher_hill("IS", key)
'UIKY'
```

`sympy.crypto.crypto.encipher_bifid(msg, key, symbols=None)`

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses an $n \times n$ Polybius square.

INPUT:

`msg`: plaintext string

`key`: short string for key; duplicate characters are ignored and then it is padded with the characters in `symbols` that were not in the short key

`symbols`: $n \times n$ characters defining the alphabet (default is `string.printable`)

OUTPUT:

ciphertext (using Bifid5 cipher without spaces)

See also:

[decipher_bifid](#) (page 344), [encipher_bifid5](#) (page 346), [encipher_bifid6](#) (page 348)

`sympy.crypto.crypto.decipher_bifid(msg, key, symbols=None)`

Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.

This is the version of the Bifid cipher that uses the $n \times n$ Polybius square.

INPUT:

`msg`: ciphertext string

`key`: short string for key; duplicate characters are ignored and then it is padded with the characters in `symbols` that were not in the short key

`symbols`: $n \times n$ characters defining the alphabet (default=`string.printable`, a 10×10 matrix)

OUTPUT:

deciphered text

Examples

```
>>> from sympy.crypto.crypto import (
...     encipher_bifid, decipher_bifid, AZ)
```

Do an encryption using the bifid5 alphabet:

```
>>> alp = AZ().replace('J', '')
>>> ct = AZ("meet me on monday!")
>>> key = AZ("gold bug")
>>> encipher_bifid(ct, key, alp)
'IEILHHFSTSFQYE'
```

When entering the text or ciphertext, spaces are ignored so it can be formatted as desired. Re-entering the ciphertext from the preceding, putting 4 characters per line and padding with an extra J, does not cause problems for the deciphering:

```
>>> decipher_bifid(''
... IEILH
... HFSTS
... FQYEJ'', key, alp)
'MEETMEONMONDAY'
```

When no alphabet is given, all 100 printable characters will be used:

```
>>> key = ''
>>> encipher_bifid('hello world!', key)
'bmtwmg-bIo*w'
>>> decipher_bifid(_, key)
'hello world!'
```

If the key is changed, a different encryption is obtained:

```
>>> key = 'gold bug'
>>> encipher_bifid('hello world!', 'gold_bug')
'hg2sfuei7t}w'
```

And if the key used to decrypt the message is not exact, the original text will not be perfectly obtained:

```
>>> decipher_bifid(_, 'gold pug')
'heldo~wor6d!'
```

`sympy.crypto.crypto.bifid5_square(key=None)`
5x5 Polybius square.

Produce the Polybius square for the 5×5 Bifid cipher.

Examples

```
>>> from sympy.crypto.crypto import bifid5_square
>>> bifid5_square("gold bug")
Matrix([
 [G, O, L, D, B],
 [U, A, C, E, F],
 [H, I, K, M, N],
 [P, Q, R, S, T],
 [V, W, X, Y, Z]])
```

`sympy.crypto.crypto.encipher_bifid5(msg, key)`
Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the 5×5 Polybius square. The letter “J” is ignored so it must be replaced with something else (traditionally an “I”) before encryption.

See also:

[decipher_bifid5](#) (page 347), [encipher_bifid](#) (page 344)

Notes

The Bifid cipher was invented around 1901 by Felix Delastelle. It is a fractional substitution cipher, where letters are replaced by pairs of symbols from a smaller alphabet. The cipher uses a 5×5 square filled with some ordering of the alphabet, except that “J” is replaced with “I” (this is a so-called Polybius square; there is a 6×6 analog if you add back in “J” and also append onto the usual 26 letter alphabet, the digits 0, 1, ..., 9). According to Helen Gaines’ book *Cryptanalysis*, this type of cipher was used in the field by the German Army during World War I.

ALGORITHM: (5x5 case)

INPUT:

`msg`: plaintext string; converted to upper case and filtered of anything but all letters except J.

`key`: short string for key; non-alphabetic letters, J and duplicated characters are ignored and then, if the length is less than 25 characters, it is padded with other letters of the alphabet (in alphabetical order).

OUTPUT:

ciphertext (all caps, no spaces)

STEPS:

0. Create the 5×5 Polybius square `S` associated to `key` as follows:

- (a) moving from left-to-right, top-to-bottom, place the letters of the key into a 5×5 matrix,
 - (b) if the key has less than 25 letters, add the letters of the alphabet not in the key until the 5×5 square is filled.
1. Create a list P of pairs of numbers which are the coordinates in the Polybius square of the letters in msg.
 2. Let L1 be the list of all first coordinates of P (length of L1 = n), let L2 be the list of all second coordinates of P (so the length of L2 is also n).
 3. Let L be the concatenation of L1 and L2 (length L = 2*n), except that consecutive numbers are paired (L[2*i], L[2*i + 1]). You can regard L as a list of pairs of length n.
 4. Let C be the list of all letters which are of the form S[i, j], for all (i, j) in L. As a string, this is the ciphertext of msg.

Examples

```
>>> from sympy.crypto.crypto import (
...     encipher_bifid5, decipher_bifid5)
```

"J" will be omitted unless it is replaced with something else:

```
>>> round_trip = lambda m, k: \
...     decipher_bifid5(encipher_bifid5(m, k), k)
>>> key = 'a'
>>> msg = "JOSIE"
>>> round_trip(msg, key)
'OSIE'
>>> round_trip(msg.replace("J", "I"), key)
'IOSIE'
>>> j = "QIQ"
>>> round_trip(msg.replace("J", j), key).replace(j, "J")
'JOSIE'
```

`sympy.crypto.crypto.decipher_bifid5(msg, key)`

Return the Bifid cipher decryption of msg.

This is the version of the Bifid cipher that uses the 5×5 Polybius square; the letter "J" is ignored unless a key of length 25 is used.

INPUT:

msg: ciphertext string

key: short string for key; duplicated characters are ignored and if the length is less than 25 characters, it will be padded with other letters from the alphabet omitting "J". Non-alphabetic characters are ignored.

OUTPUT:

plaintext from Bifid5 cipher (all caps, no spaces)

Examples

```
>>> from sympy.crypto.crypto import encipher_bifid5, decipher_bifid5
>>> key = "gold bug"
>>> encipher_bifid5('meet me on friday', key)
'IEILEHFSTSFXEE'
>>> encipher_bifid5('meet me on monday', key)
'IEILHHFSTSFQYE'
>>> decipher_bifid5(_, key)
'MEETMEONMONDAY'
```

`sympy.crypto.crypto.bifid5_square(key=None)`

5x5 Polybius square.

Produce the Polybius square for the 5×5 Bifid cipher.

Examples

```
>>> from sympy.crypto.crypto import bifid5_square
>>> bifid5_square("gold bug")
Matrix([
[G, O, L, D, B],
[U, A, C, E, F],
[H, I, K, M, N],
[P, Q, R, S, T],
[V, W, X, Y, Z]])
```

`sympy.crypto.crypto.encipher_bifid6(msg, key)`

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the 6×6 Polybius square.

INPUT:

`msg`: plaintext string (digits okay)

`key`: short string for key (digits okay). If key is less than 36 characters long, the square will be filled with letters A through Z and digits 0 through 9.

OUTPUT:

ciphertext from Bifid cipher (all caps, no spaces)

See also:

[decipher_bifid6](#) (page 348), [encipher_bifid](#) (page 344)

`sympy.crypto.crypto.decipher_bifid6(msg, key)`

Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.

This is the version of the Bifid cipher that uses the 6×6 Polybius square.

INPUT:

`msg`: ciphertext string (digits okay); converted to upper case

`key`: short string for key (digits okay). If key is less than 36 characters long, the square will be filled with letters A through Z and digits 0 through 9. All letters are converted to uppercase.

OUTPUT:

plaintext from Bifid cipher (all caps, no spaces)

Examples

```
>>> from sympy.crypto.crypto import encipher_bifid6, decipher_bifid6
>>> key = "gold bug"
>>> encipher_bifid6('meet me on monday at 8am', key)
'KFKLJJHF5MMMKTFRGPL'
>>> decipher_bifid6(_, key)
'MEETMEONMONDAYAT8AM'
```

`sympy.crypto.crypto.bifid6_square(key=None)`
6x6 Polybius square.

Produces the Polybius square for the 6×6 Bifid cipher. Assumes alphabet of symbols is "A", ..., "Z", "0", ..., "9".

Examples

```
>>> from sympy.crypto.crypto import bifid6_square
>>> key = "gold bug"
>>> bifid6_square(key)
Matrix([
[G, O, L, D, B, U],
[A, C, E, F, H, I],
[J, K, M, N, P, Q],
[R, S, T, V, W, X],
[Y, Z, 0, 1, 2, 3],
[4, 5, 6, 7, 8, 9]])
```

`sympy.crypto.crypto.rsa_public_key(p, q, e)`
Return the RSA public key pair, (n, e) , where n is a product of two primes and e is relatively prime (coprime) to the Euler totient $\phi(n)$. False is returned if any assumption is violated.

Examples

```
>>> from sympy.crypto.crypto import rsa_public_key
>>> p, q, e = 3, 5, 7
>>> rsa_public_key(p, q, e)
(15, 7)
>>> rsa_public_key(p, q, 30)
False
```

`sympy.crypto.crypto.rsa_private_key(p, q, e)`
Return the RSA private key, (n, d) , where n is a product of two primes and d is the inverse of e (mod $\phi(n)$). False is returned if any assumption is violated.

Examples

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> p, q, e = 3, 5, 7
>>> rsa_private_key(p, q, e)
(15, 7)
```

(continues on next page)

(continued from previous page)

```
>>> rsa_private_key(p, q, 30)
False
```

`sympy.crypto.crypto.encipher_rsa(i, key)`

Return encryption of i by computing $i^e \pmod n$, where key is the public key (n, e) .

Examples

```
>>> from sympy.crypto.crypto import encipher_rsa, rsa_public_key
>>> p, q, e = 3, 5, 7
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> encipher_rsa(msg, puk)
3
```

`sympy.crypto.crypto.decipher_rsa(i, key)`

Return decryption of i by computing $i^d \pmod n$, where key is the private key (n, d) .

Examples

```
>>> from sympy.crypto.crypto import decipher_rsa, rsa_private_key
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> msg = 3
>>> decipher_rsa(msg, prk)
12
```

`sympy.crypto.crypto.kid_rsa_public_key(a, b, A, B)`

Kid RSA is a version of RSA useful to teach grade school children since it does not involve exponentiation.

Alice wants to talk to Bob. Bob generates keys as follows. Key generation:

- Select positive integers a, b, A, B at random.
- Compute $M = ab - 1$, $e = AM + a$, $d = BM + b$, $n = (ed - 1)/M$.
- The public key is (n, e) . Bob sends these to Alice.
- The private key is (n, d) , which Bob keeps secret.

Encryption: If p is the plaintext message then the ciphertext is $c = pe \pmod n$.

Decryption: If c is the ciphertext message then the plaintext is $p = cd \pmod n$.

Examples

```
>>> from sympy.crypto.crypto import kid_rsa_public_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_public_key(a, b, A, B)
(369, 58)
```

`sympy.crypto.crypto.kid_rsa_private_key(a, b, A, B)`

Compute $M = ab - 1$, $e = AM + a$, $d = BM + b$, $n = (ed - 1)/M$. The private key is d , which Bob keeps secret.

Examples

```
>>> from sympy.crypto.crypto import kid_rsa_private_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_private_key(a, b, A, B)
(369, 70)
```

`sympy.crypto.crypto.encipher_kid_rsa(msg, key)`
Here `msg` is the plaintext and `key` is the public key.

Examples

```
>>> from sympy.crypto.crypto import (
...     encipher_kid_rsa, kid_rsa_public_key)
>>> msg = 200
>>> a, b, A, B = 3, 4, 5, 6
>>> key = kid_rsa_public_key(a, b, A, B)
>>> encipher_kid_rsa(msg, key)
161
```

`sympy.crypto.crypto.decipher_kid_rsa(msg, key)`
Here `msg` is the plaintext and `key` is the private key.

Examples

```
>>> from sympy.crypto.crypto import (
...     kid_rsa_public_key, kid_rsa_private_key,
...     decipher_kid_rsa, encipher_kid_rsa)
>>> a, b, A, B = 3, 4, 5, 6
>>> d = kid_rsa_private_key(a, b, A, B)
>>> msg = 200
>>> pub = kid_rsa_public_key(a, b, A, B)
>>> pri = kid_rsa_private_key(a, b, A, B)
>>> ct = encipher_kid_rsa(msg, pub)
>>> decipher_kid_rsa(ct, pri)
200
```

`sympy.crypto.crypto.encode_morse(msg, sep='|', mapping=None)`
Encodes a plaintext into popular Morse Code with letters separated by *sep* and words by a double *sep*.

References

[R81] (page 1857)

Examples

```
>>> from sympy.crypto.crypto import encode_morse
>>> msg = 'ATTACK RIGHT FLANK'
>>> encode_morse(msg)
'.-|-|-|.-|-...|-.-||.-..|...|-||...|-...|-.-|.-.-|.-.-|'
```

`sympy.crypto.crypto.decode_morse(msg, sep='|', mapping=None)`
Decodes a Morse Code with letters separated by *sep* (default is '|') and words by *wordsep* (default is '||') into plaintext.

References

[R82] (page 1857)

Examples

```
>>> from sympy.crypto.crypto import decode_morse
>>> mc = '.--|---|...-|.|||.|-|...|.-'
>>> decode_morse(mc)
'MOVE EAST'
```

`sympy.crypto.crypto.lfsr_sequence(key, fill, n)`
This function creates an lfsr sequence.

INPUT:

key: a list of finite field elements, $[c_0, c_1, \dots, c_k]$.

fill: the list of the initial terms of the lfsr sequence, $[x_0, x_1, \dots, x_k]$.

n: number of terms of the sequence that the function returns.

OUTPUT:

The lfsr sequence defined by $x_{n+1} = c_k x_n + \dots + c_0 x_{n-k}$, for $n \leq k$.

Notes

S. Golomb [G82] (page 1857) gives a list of three statistical properties a sequence of numbers $a = \{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$, should display to be considered “random”. Define the autocorrelation of a to be

$$C(k) = C(k, a) = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=1}^N (-1)^{a_n + a_{n+k}}.$$

In the case where a is periodic with period P then this reduces to

$$C(k) = \frac{1}{P} \sum_{n=1}^P (-1)^{a_n + a_{n+k}}.$$

Assume a is periodic with period P .

- balance:

$$\left| \sum_{n=1}^P (-1)^{a_n} \right| \leq 1.$$

- low autocorrelation:

$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases}$$

(For sequences satisfying these first two properties, it is known that $\epsilon = -1/P$ must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1's as there are of 0's.

References

[G82] (page 1857)

Examples

```
>>> from sympy.crypto.crypto import lfsr_sequence
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> lfsr_sequence(key, fill, 10)
[1 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 0 mod 2,
1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2]
```

`sympy.crypto.crypto.lfsr_autocorrelation(L, P, k)`

This function computes the LFSR autocorrelation function.

INPUT:

L: is a periodic sequence of elements of $GF(2)$. L must have length larger than P.

P: the period of L

k: an integer ($0 < k < p$)

OUTPUT:

the k-th value of the autocorrelation of the LFSR L

Examples

```
>>> from sympy.crypto.crypto import (
...     lfsr_sequence, lfsr_autocorrelation)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_autocorrelation(s, 15, 7)
-1/15
>>> lfsr_autocorrelation(s, 15, 0)
1
```

`sympy.crypto.crypto.lfsr_connection_polynomial(s)`

This function computes the LFSR connection polynomial.

INPUT:

`s`: a sequence of elements of even length, with entries in a finite field

OUTPUT:

$C(x)$: the connection polynomial of a minimal LFSR yielding `s`.

This implements the algorithm in section 3 of J. L. Massey's article [M83] (page 1857).

References

[M83] (page 1857)

Examples

```
>>> from sympy.crypto.crypto import (
...     lfsr_sequence, lfsr_connection_polynomial)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**4 + x + 1
>>> fill = [F(1), F(0), F(0), F(1)]
>>> key = [F(1), F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(1), F(0)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x**2 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x + 1
```

`sympy.crypto.crypto.elgamal_public_key(key)`

Return three number tuple as public key.

Parameters `key`: Tuple (p, r, e) generated by `elgamal_private_key`

Returns (p, r, e = $r^{**}d \bmod p$): d is a random number in private key.

Examples

```
>>> from sympy.crypto.crypto import elgamal_public_key
>>> elgamal_public_key((1031, 14, 636))
(1031, 14, 212)
```


`sympy.crypto.crypto.elgamal_private_key(digit=10, seed=None)`

Return three number tuple as private key.

Elgamal encryption is based on the mathematical problem called the Discrete Logarithm Problem (DLP). For example,

$$a^b \equiv c \pmod{p}$$

In general, if a and b are known, ct is easily calculated. If b is unknown, it is hard to use a and ct to get b .

Parameters **digit** : minimum number of binary digits for key

Returns **(p, r, d)** : p = prime number, r = primitive root, d = random number

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.utilities.randtest._randrange`.

Examples

```
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.ntheory import is_primitive_root, isprime
>>> a, b, _ = elgamal_private_key()
>>> isprime(a)
True
>>> is_primitive_root(b, a)
True
```

`sympy.crypto.crypto.encipher_elgamal(i, key, seed=None)`

Encrypt message with public key

i is a plaintext message expressed as an integer. key is public key (p, r, e) . In order to encrypt a message, a random number a in $\text{range}(2, p)$ is generated and the encrypted message is returned as c_1 and c_2 where:

$$c_1 \equiv r^a \pmod{p}$$

$$c_2 \equiv me^a \pmod{p}$$

Parameters **msg** : int of encoded message

key : public key

Returns **(c1, c2)** : Encipher into two number

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.utilities.randtest._randrange`.

Examples

```
>>> from sympy.crypto.crypto import encipher_elgamal, elgamal_private_key, e
    elgamal_public_key
>>> pri = elgamal_private_key(5, seed=[3]); pri
(37, 2, 3)
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 36
>>> encipher_elgamal(msg, pub, seed=[3])
(8, 6)
```

`sympy.crypto.crypto.decipher_elgamal(msg, key)`

Decrypt message with private key

$msg = (c_1, c_2)$

$key = (p, r, d)$

According to extended Eucliden theorem, $uc_1^d + pn = 1$

$u \equiv 1/c_1^d \pmod{p}$

$uc_2 \equiv \frac{1}{c_1^d} c_2 \equiv \frac{1}{r^{ad}} c_2 \pmod{p}$

$\frac{1}{r^{ad}} me^a \equiv \frac{1}{r^{ad}} mr^{da} \equiv m \pmod{p}$

Examples

```
>>> from sympy.crypto.crypto import decipher_elgamal
>>> from sympy.crypto.crypto import encipher_elgamal
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.crypto.crypto import elgamal_public_key
```

```
>>> pri = elgamal_private_key(5, seed=[3])
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 17
>>> decipher_elgamal(encipher_elgamal(msg, pub), pri) == msg
True
```

`sympy.crypto.crypto.dh_public_key(key)`

Return three number tuple as public key.

This is the tuple that Alice sends to Bob.

Parameters *key*: Tuple (p, g, a) generated by “dh_private_key”

Returns (p, g, g^a mod p) : p, g and a as in Parameters

Examples

```
>>> from sympy.crypto.crypto import dh_private_key, dh_public_key
>>> p, g, a = dh_private_key();
>>> _p, _g, x = dh_public_key((p, g, a))
>>> p == _p and g == _g
True
>>> x == pow(g, a, p)
True
```

`sympy.crypto.crypto.dh_private_key(digit=10, seed=None)`

Return three integer tuple as private key.

Diffie-Hellman key exchange is based on the mathematical problem called the Discrete Logarithm Problem (see ElGamal).

Diffie-Hellman key exchange is divided into the following steps:

- Alice and Bob agree on a base that consist of a prime p and a primitive root of p called g
- Alice choses a number a and Bob choses a number b where a and b are random numbers in range $[2, p)$. These are their private keys.
- Alice then publicly sends Bob $g^a \pmod{p}$ while Bob sends Alice $g^b \pmod{p}$
- They both raise the received value to their secretly chosen number (a or b) and now have both as their shared key $g^{ab} \pmod{p}$

Parameters digit: minimum number of binary digits required in key

Returns (p, g, a) : p = prime number, g = primitive root of p ,
 a = random number from 2 through $p - 1$

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.utilities.randtest._randrange`.

Examples

```
>>> from sympy.crypto.crypto import dh_private_key
>>> from sympy.ntheory import isprime, is_primitive_root
>>> p, g, _ = dh_private_key()
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
>>> p, g, _ = dh_private_key(5)
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
```

`sympy.crypto.crypto.dh_shared_key(key, b)`

Return an integer that is the shared key.

This is what Bob and Alice can both calculate using the public keys they received from each other and their private keys.

Parameters key: Tuple (p, g, x) generated by “`dh_public_key`”

b: Random number in the range of 2 to $p - 1$

(Chosen by second key exchange member (Bob))

Returns shared key (int)

Examples

```
>>> from sympy.crypto.crypto import (  
...     dh_private_key, dh_public_key, dh_shared_key)  
>>> prk = dh_private_key();  
>>> p, g, x = dh_public_key(prk);  
>>> sk = dh_shared_key((p, g, x), 1000)  
>>> sk == pow(x, 1000, p)  
True
```

`sympy.crypto.crypto.encipher_elgamal(i, key, seed=None)`

Encrypt message with public key

i is a plaintext message expressed as an integer. *key* is public key (*p*, *r*, *e*). In order to encrypt a message, a random number *a* in `range(2, p)` is generated and the encrypted message is returned as *c*₁ and *c*₂ where:

$$c_1 \equiv r^a \pmod{p}$$

$$c_2 \equiv me^a \pmod{p}$$

Parameters *msg* : int of encoded message

key : public key

Returns (*c1*, *c2*) : Encipher into two number

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See `sympy.utilities.randtest._randrange`.

Examples

```
>>> from sympy.crypto.crypto import encipher_elgamal, elgamal_private_key,   
↪ elgamal_public_key  
>>> pri = elgamal_private_key(5, seed=[3]); pri  
(37, 2, 3)  
>>> pub = elgamal_public_key(pri); pub  
(37, 2, 8)  
>>> msg = 36  
>>> encipher_elgamal(msg, pub, seed=[3])  
(8, 6)
```

`sympy.crypto.crypto.decipher_elgamal(msg, key)`

Decrypt message with private key

$$msg = (c_1, c_2)$$

$$key = (p, r, d)$$

According to extended Eucliden theorem, $uc_1^d + pn = 1$

$$u \equiv 1/c_1^d \pmod{p}$$

$$uc_2 \equiv \frac{1}{c_1^d} c_2 \equiv \frac{1}{r^{ad}} c_2 \pmod{p}$$

$$\frac{1}{r^{ad}} me^a \equiv \frac{1}{r^{ad}} mr^{da} \equiv m \pmod{p}$$

Examples

```
>>> from sympy.crypto.crypto import decipher_elgamal
>>> from sympy.crypto.crypto import encipher_elgamal
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.crypto.crypto import elgamal_public_key
```

```
>>> pri = elgamal_private_key(5, seed=[3])
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 17
>>> decipher_elgamal(encipher_elgamal(msg, pub), pri) == msg
True
```

`sympy.crypto.crypto.gm_public_key(p, q, a=None, seed=None)`

Compute public keys for p and q. Note that in Goldwasser-Micali Encryption, public keys are randomly selected.

Parameters **p, q, a** : (int) initialization variables

Returns (**a, N**) : tuple[int]

a is the input a if it is not None otherwise some random integer coprime to p and q.

N is the product of p and q

`sympy.crypto.crypto.gm_private_key(p, q, a=None)`

Check if p and q can be used as private keys for the Goldwasser-Micali encryption. The method works roughly as follows.

Pick two large primes p and q. Call their product N. Given a message as an integer i, write i in its bit representation $b_0, \dots, b_n$. For each k,

if $b_k = 0$: let a_k be a random square (quadratic residue) modulo $p * q$ such that $\text{jacobi\_symbol}(a, p * q) = 1$

if $b_k = 1$: let a_k be a random non-square (non-quadratic residue) modulo $p * q$ such that $\text{jacobi\_symbol}(a, p * q) = 1$

return $[a_1, a_2, \dots]$

b_k can be recovered by checking whether or not a_k is a residue. And from the b_k 's, the message can be reconstructed.

The idea is that, while $\text{jacobi\_symbol}(a, p * q)$ can be easily computed (and when it is equal to -1 will tell you that a is not a square mod $p * q$), quadratic residuosity modulo a composite number is hard to compute without knowing its factorization.

Moreover, approximately half the numbers coprime to $p * q$ have jacobi_symbol equal to 1. And among those, approximately half are residues and approximately half are not. This maximizes the entropy of the code.

Parameters **p, q, a** : initialization variables

Returns **p, q** : the input value p and q

Raises **ValueError** : if p and q are not distinct odd primes

`sympy.crypto.crypto.encipher_gm(i, key, seed=None)`

Encrypt integer 'i' using public_key 'key' Note that gm uses random encryption.

Parameters **i**: (int) the message to encrypt

key: Tuple (a, N) the public key

Returns List[int] the randomized encrypted message.

`sympy.crypto.crypto.decipher_gm(message, key)`
Decrypt message 'message' using public_key 'key'.

Parameters List[int]: the randomized encrypted message.

key: Tuple (p, q) the private key

Returns i (int) the encrypted message

5.5 Concrete Mathematics

5.5.1 Hypergeometric terms

The center stage, in recurrence solving and summations, play hypergeometric terms. Formally these are sequences annihilated by first order linear recurrence operators. In simple words if we are given term $a(n)$ then it is hypergeometric if its consecutive term ratio is a rational function in n .

To check if a sequence is of this type you can use the `is_hypergeometric` method which is available in Basic class. Here is simple example involving a polynomial:

```
>>> from sympy import *
>>> n, k = symbols('n,k')
>>> (n**2 + 1).is_hypergeometric(n)
True
```

Of course polynomials are hypergeometric but are there any more complicated sequences of this type? Here are some trivial examples:

```
>>> factorial(n).is_hypergeometric(n)
True
>>> binomial(n, k).is_hypergeometric(n)
True
>>> rf(n, k).is_hypergeometric(n)
True
>>> ff(n, k).is_hypergeometric(n)
True
>>> gamma(n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

We see that all species used in summations and other parts of concrete mathematics are hypergeometric. Note also that binomial coefficients and both rising and falling factorials are hypergeometric in both their arguments:

```
>>> binomial(n, k).is_hypergeometric(k)
True
>>> rf(n, k).is_hypergeometric(k)
True
>>> ff(n, k).is_hypergeometric(k)
True
```

To say more, all previously shown examples are valid for integer linear arguments:

```
>>> factorial(2*n).is_hypergeometric(n)
True
>>> binomial(3*n+1, k).is_hypergeometric(n)
True
>>> rf(n+1, k-1).is_hypergeometric(n)
True
>>> ff(n-1, k+1).is_hypergeometric(n)
True
>>> gamma(5*n).is_hypergeometric(n)
True
>>> (2**(n-7)).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise it will return `None` to say that sequence is not hypergeometric:

```
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

5.5.2 Concrete Class Reference

class `sympy.concrete.summations.Sum`

Represents unevaluated summation.

`Sum` represents a finite or infinite series, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the summation.

See also:

`summation` (page 371), `Product`, `product`

References

[R37] (page 1857), [R38] (page 1857), [R39] (page 1857)

Examples

```
>>> from sympy.abc import i, k, m, n, x
>>> from sympy import Sum, factorial, oo, IndexedBase, Function
>>> Sum(k, (k, 1, m))
Sum(k, (k, 1, m))
```

(continues on next page)

(continued from previous page)

```

>>> Sum(k, (k, 1, m)).doit()
m**2/2 + m/2
>>> Sum(k**2, (k, 1, m))
Sum(k**2, (k, 1, m))
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k, (k, 0, oo))
Sum(x**k, (k, 0, oo))
>>> Sum(x**k, (k, 0, oo)).doit()
Piecewise((1/(-x + 1), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), True))
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()
exp(x)

```

Here are examples to do summation with symbolic indices. You can use either Function of IndexedBase classes:

```

>>> f = Function('f')
>>> Sum(f(n), (n, 0, 3)).doit()
f(0) + f(1) + f(2) + f(3)
>>> Sum(f(n), (n, 0, oo)).doit()
Sum(f(n), (n, 0, oo))
>>> f = IndexedBase('f')
>>> Sum(f[n]**2, (n, 0, 3)).doit()
f[0]**2 + f[1]**2 + f[2]**2 + f[3]**2

```

An example showing that the symbolic result of a summation is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those sums by interchanging the limits according to the above rules:

```

>>> S = Sum(i, (i, 1, n)).doit()
>>> S
n**2/2 + n/2
>>> S.subs(n, -4)
6
>>> Sum(i, (i, 1, -4)).doit()
6
>>> Sum(-i, (i, -3, 0)).doit()
6

```

An explicit example of the Karr summation convention:

```

>>> S1 = Sum(i**2, (i, m, m+n-1)).doit()
>>> S1
m**2*n + m*n**2 - m*n + n**3/3 - n**2/2 + n/6
>>> S2 = Sum(i**2, (i, m+n, m-1)).doit()
>>> S2
-m**2*n - m*n**2 + m*n - n**3/3 + n**2/2 - n/6
>>> S1 + S2
0
>>> S3 = Sum(i, (i, m, m-1)).doit()
>>> S3
0

```


Finite Sums

For finite sums (and sums with symbolic limits assumed to be finite) we follow the summation convention described by Karr [1], especially definition 3 of section 1.4. The sum:

$$\sum_{m \leq i < n} f(i)$$

has the obvious meaning for $m < n$, namely:

$$\sum_{m \leq i < n} f(i) = f(m) + f(m+1) + \dots + f(n-2) + f(n-1)$$

with the upper limit value $f(n)$ excluded. The sum over an empty set is zero if and only if $m = n$:

$$\sum_{m \leq i < n} f(i) = 0 \quad \text{for} \quad m = n$$

Finally, for all other sums over empty sets we assume the following definition:

$$\sum_{m \leq i < n} f(i) = - \sum_{n \leq i < m} f(i) \quad \text{for} \quad m > n$$

It is important to note that Karr defines all sums with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the summation convention. Indeed we have:

$$\sum_{m \leq i < n} f(i) = \sum_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

euler_maclaurin(m=0, n=0, eps=0, eval_integral=True)

Return an Euler-Maclaurin approximation of self, where m is the number of leading terms to sum directly and n is the number of terms in the tail.

With m = n = 0, this is simply the corresponding integral plus a first-order endpoint correction.

Returns (s, e) where s is the Euler-Maclaurin approximation and e is the estimated error (taken to be the magnitude of the first omitted term in the tail):

```
>>> from sympy.abc import k, a, b
>>> from sympy import Sum
>>> Sum(1/k, (k, 2, 5)).doit().evalf()
1.28333333333333
>>> s, e = Sum(1/k, (k, 2, 5)).euler_maclaurin()
>>> s
-log(2) + 7/20 + log(5)
>>> from sympy import sstr
>>> print(sstr((s.evalf(), e.evalf()), full_prec=True))
(1.26629073187415, 0.0175000000000000)
```

The endpoints may be symbolic:

```
>>> s, e = Sum(1/k, (k, a, b)).euler_maclaurin()
>>> s
-log(a) + log(b) + 1/(2*b) + 1/(2*a)
>>> e
Abs(1/(12*b**2) - 1/(12*a**2))
```

If the function is a polynomial of degree at most $2n+1$, the Euler-Maclaurin formula becomes exact (and $e = 0$ is returned):

```
>>> Sum(k, (k, 2, b)).euler_maclaurin()
(b**2/2 + b/2 - 1, 0)
>>> Sum(k, (k, 2, b)).doit()
b**2/2 + b/2 - 1
```

With a nonzero `eps` specified, the summation is ended as soon as the remainder term is less than the `epsilon`.

eval_zeta_function(f, limits)

Check whether the function matches with the zeta function. If it matches, then return a *Piecewise* expression because zeta function does not converge unless $s > 1$ and $q > 0$

is_absolutely_convergent()

Checks for the absolute convergence of an infinite series.

Same as checking convergence of absolute value of `sequence_term` of an infinite series.

See also:

[Sum.is_convergent](#) (page 364)

References

[R40] (page 1857)

Examples

```
>>> from sympy import Sum, Symbol, sin, oo
>>> n = Symbol('n', integer=True)
>>> Sum((-1)**n, (n, 1, oo)).is_absolutely_convergent()
False
>>> Sum((-1)**n/n**2, (n, 1, oo)).is_absolutely_convergent()
True
```

is_convergent()

Checks for the convergence of a `Sum`.

We divide the study of convergence of infinite sums and products in two parts.

First Part: One part is the question whether all the terms are well defined, i.e., they are finite in a sum and also non-zero in a product. Zero is the analogy of (minus) infinity in products as $e^{-\infty} = 0$.

Second Part: The second part is the question of convergence after infinities, and zeros in products, have been omitted assuming that their number is finite. This

means that we only consider the tail of the sum or product, starting from some point after which all terms are well defined.

For example, in a sum of the form:

$$\sum_{1 \leq i < \infty} \frac{1}{n^2 + an + b}$$

where a and b are numbers. The routine will return true, even if there are infinities in the term sequence (at most two). An analogous product would be:

$$\prod_{1 \leq i < \infty} e^{\frac{1}{n^2 + an + b}}$$

This is how convergence is interpreted. It is concerned with what happens at the limit. Finding the bad terms is another independent matter.

Note: It is responsibility of user to see that the sum or product is well defined.

There are various tests employed to check the convergence like divergence test, root test, integral test, alternating series test, comparison tests, Dirichlet tests. It returns true if Sum is convergent and false if divergent and NotImplementedError if it can not be checked.

See also:

[Sum.is_absolutely_convergent](#) (page 364), [Product.is_convergent](#)

References

[R41] (page 1857)

Examples

```
>>> from sympy import factorial, S, Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum(n/(n - 1), (n, 4, 7)).is_convergent()
True
>>> Sum(n/(2*n + 1), (n, 1, oo)).is_convergent()
False
>>> Sum(factorial(n)/5**n, (n, 1, oo)).is_convergent()
False
>>> Sum(1/n**(S(6)/5), (n, 1, oo)).is_convergent()
True
```

reverse_order(*indices)

Reverse the order of a limit in a Sum.

See also:

[index](#), [reorder_limit](#), [reorder](#)

References

[R42] (page 1857)

Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y, a, b, c, d
```

```
>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x**2, (x, a, b), (x, c, d))
>>> S
Sum(x**2, (x, a, b), (x, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

Usage

`reverse_order(self, *indices)` reverses some limits in the expression `self` which can be either a `Sum` or a `Product`. The selectors in the argument `indices` specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

class sympy.concrete.products.Product

Represents unevaluated products.

`Product` represents a finite or infinite product, with the first argument being the general form of terms in the series, and the second argument being (`dummy_variable`, `start`, `end`), with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the product.

See also:

`Sum`, `summation`, [product](#) (page 372)

References

[R43] (page 1857), [R44] (page 1857), [R45] (page 1857)

Examples

```
>>> from sympy.abc import a, b, i, k, m, n, x
>>> from sympy import Product, factorial, oo
>>> Product(k, (k, 1, m))
Product(k, (k, 1, m))
>>> Product(k, (k, 1, m)).doit()
factorial(m)
>>> Product(k**2, (k, 1, m))
Product(k**2, (k, 1, m))
>>> Product(k**2, (k, 1, m)).doit()
factorial(m)**2
```

Wallis' product for pi:

```
>>> W = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, oo))
>>> W
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

Direct computation currently fails:

```
>>> W.doit()
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

But we can approach the infinite product by a limit of finite products:

```
>>> from sympy import limit
>>> W2 = Product(2*i/(2*i-1)*2*i/(2*i+1), (i, 1, n))
>>> W2
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, n))
>>> W2e = W2.doit()
>>> W2e
2**(-2*n)*4**n*factorial(n)**2/(RisingFactorial(1/2, n)*RisingFactorial(3/2, n))
>>> limit(W2e, n, oo)
pi/2
```

By the same formula we can compute $\sin(\pi/2)$:

```
>>> from sympy import pi, gamma, simplify
>>> P = pi * x * Product(1 - x**2/k**2, (k, 1, n))
>>> P = P.subs(x, pi/2)
>>> P
pi**2*Product(1 - pi**2/(4*k**2), (k, 1, n))/2
>>> Pe = P.doit()
>>> Pe
pi**2*RisingFactorial(1 + pi/2, n)*RisingFactorial(-pi/2 + 1, n)/
↳ (2*factorial(n)**2)
>>> Pe = Pe.rewrite(gamma)
>>> Pe
pi**2*gamma(n + 1 + pi/2)*gamma(n - pi/2 + 1)/(2*gamma(1 + pi/2)*gamma(-pi/2 +
↳ 1)*gamma(n + 1)**2)
>>> Pe = simplify(Pe)
```

(continues on next page)

(continued from previous page)

```
>>> Pe
sin(pi**2/2)*gamma(n + 1 + pi/2)*gamma(n - pi/2 + 1)/gamma(n + 1)**2
>>> limit(Pe, n, oo)
sin(pi**2/2)
```

Products with the lower limit being larger than the upper one:

```
>>> Product(1/i, (i, 6, 1)).doit()
120
>>> Product(i, (i, 2, 5)).doit()
120
```

The empty product:

```
>>> Product(i, (i, n, n-1)).doit()
1
```

An example showing that the symbolic result of a product is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those products by interchanging the limits according to the above rules:

```
>>> P = Product(2, (i, 10, n)).doit()
>>> P
2**(n - 9)
>>> P.subs(n, 5)
1/16
>>> Product(2, (i, 10, 5)).doit()
1/16
>>> 1/Product(2, (i, 6, 9)).doit()
1/16
```

An explicit example of the Karr summation convention applied to products:

```
>>> P1 = Product(x, (i, a, b)).doit()
>>> P1
x**(-a + b + 1)
>>> P2 = Product(x, (i, b+1, a-1)).doit()
>>> P2
x**(a - b - 1)
>>> simplify(P1 * P2)
1
```

And another one:

```
>>> P1 = Product(i, (i, b, a)).doit()
>>> P1
RisingFactorial(b, a - b + 1)
>>> P2 = Product(i, (i, a+1, b-1)).doit()
>>> P2
RisingFactorial(a + 1, -a + b - 1)
>>> P1 * P2
RisingFactorial(b, a - b + 1)*RisingFactorial(a + 1, -a + b - 1)
>>> simplify(P1 * P2)
1
```

Finite Products

For finite products (and products with symbolic limits assumed to be finite) we follow the analogue of the summation convention described by Karr [1], especially definition 3 of section 1.4. The product:

$$\prod_{m \leq i < n} f(i)$$

has the obvious meaning for $m < n$, namely:

$$\prod_{m \leq i < n} f(i) = f(m)f(m+1) \cdot \dots \cdot f(n-2)f(n-1)$$

with the upper limit value $f(n)$ excluded. The product over an empty set is one if and only if $m = n$:

$$\prod_{m \leq i < n} f(i) = 1 \quad \text{for } m = n$$

Finally, for all other products over empty sets we assume the following definition:

$$\prod_{m \leq i < n} f(i) = \frac{1}{\prod_{n \leq i < m} f(i)} \quad \text{for } m > n$$

It is important to note that above we define all products with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the product convention. Indeed we have:

$$\prod_{m \leq i < n} f(i) = \prod_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

`is_convergent()`

See docs of `Sum.is_convergent()` for explanation of convergence in SymPy.

The infinite product:

$$\prod_{1 \leq i < \infty} f(i)$$

is defined by the sequence of partial products:

$$\prod_{i=1}^n f(i) = f(1)f(2) \cdots f(n)$$

as n increases without bound. The product converges to a non-zero value if and only if the sum:

$$\sum_{1 \leq i < \infty} \log f(i)$$

converges.

References

[R46] (page 1857)

Examples

```
>>> from sympy import Interval, S, Product, Symbol, cos, pi, exp, oo
>>> n = Symbol('n', integer=True)
>>> Product(n/(n + 1), (n, 1, oo)).is_convergent()
False
>>> Product(1/n**2, (n, 1, oo)).is_convergent()
False
>>> Product(cos(pi/n), (n, 1, oo)).is_convergent()
True
>>> Product(exp(-n**2), (n, 1, oo)).is_convergent()
False
```

reverse_order(*indices)

Reverse the order of a limit in a Product.

See also:

index, reorder_limit, reorder

References

[R47] (page 1857)

Examples

```
>>> from sympy import Product, simplify, RisingFactorial, gamma, Sum
>>> from sympy.abc import x, y, a, b, c, d
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr)
gamma(b + 1)/gamma(a)
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P)
gamma(b + 1)/gamma(a)
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x*y, (x, a, b), (y, c, d))
>>> S
Sum(x*y, (x, a, b), (y, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> S1 = S0.reverse_order(1)
```

(continues on next page)

(continued from previous page)

```
>>> S1
Sum(x*y, (x, b + 1, a - 1), (y, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

Usage

`reverse_order(expr, *indices)` reverses some limits in the expression `expr` which can be either a `Sum` or a `Product`. The selectors in the argument `indices` specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

5.5.3 Concrete Functions Reference

`sympy.concrete.summations.summation(f, *symbols, **kwargs)`

Compute the summation of `f` with respect to `symbols`.

The notation for symbols is similar to the notation used in `Integral`. `summation(f, (i, a, b))` computes the sum of `f` with respect to `i` from `a` to `b`, i.e.,

$$\text{summation}(f, (i, a, b)) = \sum_{i=a}^b f$$

If it cannot compute the sum, it returns an unevaluated `Sum` object. Repeated sums can be computed by introducing additional symbols tuples:

```
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)
```

```
>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2
>>> summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
>>> summation(i, (i, 0, n), (n, 0, m))
m**3/6 + m**2/2 + m/3
```

```
>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
exp(x)
```

See also:

[Sum](#) (page 361), [Product](#), [product](#)

`sympy.concrete.products.product(*args, **kwargs)`

Compute the product.

The notation for symbols is similar to the notation used in `Sum` or `Integral`. `product(f, (i, a, b))` computes the product of `f` with respect to `i` from `a` to `b`, i.e.,

$$\text{product}(f(n), (i, a, b)) = \prod_{i=a}^b f(n)$$

If it cannot compute the product, it returns an unevaluated `Product` object. Repeated products can be computed by introducing additional symbols tuples:

```
>>> from sympy import product, symbols
>>> i, n, m, k = symbols('i n m k', integer=True)
```

```
>>> product(i, (i, 1, k))
factorial(k)
>>> product(m, (i, 1, k))
m**k
>>> product(i, (i, 1, k), (k, 1, n))
Product(factorial(k), (k, 1, n))
```

`sympy.concrete.gosper.gosper_normal(f, g, n, polys=True)`

Compute the Gosper's normal form of `f` and `g`.

Given relatively prime univariate polynomials `f` and `g`, rewrite their quotient to a normal form defined as follows:

$$\frac{f(n)}{g(n)} = Z \cdot \frac{A(n)C(n+1)}{B(n)C(n)}$$

where `Z` is an arbitrary constant and `A`, `B`, `C` are monic polynomials in `n` with the following properties:

1. $\gcd(A(n), B(n+h)) = 1 \forall h \in \mathbb{N}$
2. $\gcd(B(n), C(n+1)) = 1$
3. $\gcd(A(n), C(n)) = 1$

This normal form, or rational factorization in other words, is a crucial step in Gosper's algorithm and in solving of difference equations. It can be also used to decide if two hypergeometric terms are similar or not.

This procedure will return a tuple containing elements of this factorization in the form `(Z*A, B, C)`.

Examples

```
>>> from sympy.concrete.gosper import gosper_normal
>>> from sympy.abc import n
```

```
>>> gosper_normal(4*n+5, 2*(4*n+1)*(2*n+3), n, polys=False)
(1/4, n + 3/2, n + 1/4)
```

`sympy.concrete.gosper.gosper_term(f, n)`
 Compute Gosper's hypergeometric term for f .
 Suppose f is a hypergeometric term such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and f_k doesn't depend on n . Returns a hypergeometric term g_n such that $g_{n+1} - g_n = f_n$.

Examples

```
>>> from sympy.concrete.gosper import gosper_term
>>> from sympy.functions import factorial
>>> from sympy.abc import n
```

```
>>> gosper_term((4*n + 1)*factorial(n)/factorial(2*n + 1), n)
(-n - 1/2)/(n + 1/4)
```

`sympy.concrete.gosper.gosper_sum(f, k)`
 Gosper's hypergeometric summation algorithm.
 Given a hypergeometric term f such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and $f(n)$ doesn't depend on n , returns $g_n - g(0)$ where $g_{n+1} - g_n = f_n$, or None if s_n can not be expressed in closed form as a sum of hypergeometric terms.

References

[R48] (page 1857)

Examples

```
>>> from sympy.concrete.gosper import gosper_sum
>>> from sympy.functions import factorial
>>> from sympy.abc import i, n, k
```

```
>>> f = (4*k + 1)*factorial(k)/factorial(2*k + 1)
>>> gosper_sum(f, (k, 0, n))
(-factorial(n) + 2*factorial(2*n + 1))/factorial(2*n + 1)
>>> _.subs(n, 2) == sum(f.subs(k, i) for i in [0, 1, 2])
True
>>> gosper_sum(f, (k, 3, n))
(-60*factorial(n) + factorial(2*n + 1))/(60*factorial(2*n + 1))
>>> _.subs(n, 5) == sum(f.subs(k, i) for i in [3, 4, 5])
True
```

5.6 Numerical evaluation

5.6.1 Basics

Exact SymPy expressions can be converted to floating-point approximations (decimal numbers) using either the `.evalf()` method or the `N()` function. `N(expr, <args>)` is equivalent to `sympify(expr).evalf(<args>)`.

```
>>> from sympy import *
>>> N(sqrt(2)*pi)
4.44288293815837
>>> (sqrt(2)*pi).evalf()
4.44288293815837
```

By default, numerical evaluation is performed to an accuracy of 15 decimal digits. You can optionally pass a desired accuracy (which should be a positive integer) as an argument to `evalf` or `N`:

```
>>> N(sqrt(2)*pi, 5)
4.4429
>>> N(sqrt(2)*pi, 50)
4.4428829381583662470158809900606936986146216893757
```

Complex numbers are supported:

```
>>> N(1/(pi + I), 20)
0.2890254822223624241 - 0.091999668350375232456*I
```

If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `.evalf()` or `N()` returns the original expression, or in some cases a partially evaluated expression. For example, when the expression is a polynomial in expanded form, the coefficients are evaluated:

```
>>> x = Symbol('x')
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x**2 + 0.333333333333333*x
```

You can also use the standard Python functions `float()`, `complex()` to convert SymPy expressions to regular Python numbers:

```
>>> float(pi)
3.1415926535...
>>> complex(pi+E*I)
(3.1415926535...+2.7182818284...j)
```

If these functions are used, failure to evaluate the expression to an explicit number (for example if the expression contains symbols) will raise an exception.

There is essentially no upper precision limit. The following command, for example, computes the first 100,000 digits of π/e :

```
>>> N(pi/E, 100000)
...
```

This shows digits 999,951 through 1,000,000 of π :

High-precision calculations can be slow. It is recommended (but entirely optional) to install gmpy (<http://code.google.com/p/gmpy/>), which will significantly speed up computations such as the one above.

As the last example shows, some Python floats are only accurate to about 15 digits as inputs, while others (those that have a denominator that is a power of 2, like $0.125 = 1/8$) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:

The precision of a number determines 1) the precision to use when performing arithmetic with the number, and 2) the number of digits to display when printing the number. When two numbers with different precision are used together in an arithmetic operation, the higher of the precisions is used for the result. The product of 0.1 ± 0.001 and 3.1415 ± 0.0001 has an uncertainty of about 0.003 and yet 5 digits of precision are shown.

So the displayed precision should not be used as a model of error propagation or significance arithmetic; rather, this scheme is employed to ensure stability of numerical algorithms.

5.6. Numerical evaluation 375

5.6.3 Accuracy and error handling

When the input to `N` or `evalf` is a complicated expression, numerical error propagation becomes a concern. As an example, consider the 100'th Fibonacci number and the excellent (but not exact) approximation $\varphi^{100}/\sqrt{5}$ where φ is the golden ratio. With ordinary floating-point arithmetic, subtracting these numbers from each other erroneously results in a complete cancellation:

```
>>> a, b = GoldenRatio**1000/sqrt(5), fibonacci(1000)
>>> float(a)
4.34665576869e+208
>>> float(b)
4.34665576869e+208
>>> float(a) - float(b)
0.0
```

`N` and `evalf` keep track of errors and automatically increase the precision used internally in order to obtain a correct result:

```
>>> N(fibonacci(100) - GoldenRatio**100/sqrt(5))
-5.64613129282185e-22
```

Unfortunately, numerical evaluation cannot tell an expression that is exactly zero apart from one that is merely very small. The working precision is therefore capped, by default to around 100 digits. If we try with the 1000'th Fibonacci number, the following happens:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5))
0.e+85
```

The lack of digits in the returned number indicates that `N` failed to achieve full accuracy. The result indicates that the magnitude of the expression is something less than 10^{84} , but that is not a particularly good answer. To force a higher working precision, the `maxn` keyword argument can be used:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), maxn=500)
-4.60123853010113e-210
```

Normally, `maxn` can be set very high (thousands of digits), but be aware that this may cause significant slowdown in extreme cases. Alternatively, the `strict=True` option can be set to force an exception instead of silently returning a value with less than the requested accuracy:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), strict=True)
Traceback (most recent call last):
...
PrecisionExhausted: Failed to distinguish the expression:
-sqrt(5)*GoldenRatio**1000/5 + 
↪ 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255
from zero. Try simplifying the input, using chop=True, or providing a higher maxn for 
↪ evalf
```

If we add a term so that the Fibonacci approximation becomes exact (the full form of Binet's formula), we get an expression that is exactly zero, but `N` does not know this:

```
>>> f = fibonacci(100) - (GoldenRatio**100 - (GoldenRatio-1)**100)/sqrt(5)
>>> N(f)
```

(continues on next page)

(continued from previous page)

```
0.e-104
>>> N(f, maxn=1000)
0.e-1336
```

In situations where such cancellations are known to occur, the chop options is useful. This basically replaces very small numbers in the real or imaginary portions of a number with exact zeros:

```
>>> N(f, chop=True)
0
>>> N(3 + I*f, chop=True)
3.000000000000000
```

In situations where you wish to remove meaningless digits, re-evaluation or the use of the round method are useful:

```
>>> Float('.1', '')*Float('.12345', '')
0.012297
>>> ans = _
>>> N(ans, 1)
0.01
>>> ans.round(2)
0.01
```

If you are dealing with a numeric expression that contains no floats, it can be evaluated to arbitrary precision. To round the result relative to a given decimal, the round method is useful:

```
>>> v = 10*pi + cos(1)
>>> N(v)
31.9562288417661
>>> v.round(3)
31.956
```

5.6.4 Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```
>>> var('n x')
(n, x)
>>> Sum(1/n**n, (n, 1, oo)).evalf()
1.29128599706266
>>> Integral(x**(-x), (x, 0, 1)).evalf()
1.29128599706266
>>> Sum(1/n**n, (n, 1, oo)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> Integral(x**(-x), (x, 0, 1)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> (Integral(exp(-x**2), (x, -oo, oo)) ** 2).evalf(30)
3.14159265358979323846264338328
```

By default, the tanh-sinh quadrature algorithm is used to evaluate integrals. This algorithm is very efficient and robust for smooth integrands (and even integrals with endpoint singularities), but may struggle with integrals that are highly oscillatory or have mid-interval disconti-

nities. In many cases, `evalf/N` will correctly estimate the error. With the following integral, the result is accurate but only good to four digits:

```
>>> f = abs(sin(x))
>>> Integral(abs(sin(x)), (x, 0, 4)).evalf()
2.346
```

It is better to split this integral into two pieces:

```
>>> (Integral(f, (x, 0, pi)) + Integral(f, (x, pi, 4))).evalf()
2.34635637913639
```

A similar example is the following oscillatory integral:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(maxn=20)
0.5
```

It can be dealt with much more efficiently by telling `evalf` or `N` to use an oscillatory quadrature algorithm:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(quad='osc')
0.504067061906928
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(20, quad='osc')
0.50406706190692837199
```

Oscillatory quadrature requires an integrand containing a factor $\cos(ax+b)$ or $\sin(ax+b)$. Note that many other oscillatory integrals can be transformed to this form with a change of variables:

```
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> intgr1 = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgr1
oo
/
|
| sin(x)
| ----- dx
|      2
|      x
|
/
1
>>> N(intgr1, quad='osc')
0.504067061906928
```

Infinite series use direct summation if the series converges quickly enough. Otherwise, extrapolation methods (generally the Euler-Maclaurin formula but also Richardson extrapolation) are used to speed up convergence. This allows high-precision evaluation of slowly convergent series:

```
>>> var('k')
k
>>> Sum(1/k**2, (k, 1, oo)).evalf()
1.64493406684823
>>> zeta(2).evalf()
1.64493406684823
>>> Sum(1/k - log(1+1/k), (k, 1, oo)).evalf()
0.577215664901533
```

(continues on next page)

(continued from previous page)

```
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf(50)
0.57721566490153286060651209008240243104215933593992
>>> EulerGamma.evalf(50)
0.57721566490153286060651209008240243104215933593992
```

The Euler-Maclaurin formula is also used for finite series, allowing them to be approximated quickly without evaluating all terms:

```
>>> Sum(1/k, (k, 10000000, 20000000)).evalf()
0.693147255559946
```

Note that `evalf` makes some assumptions that are not always optimal. For fine-tuned control over numerical summation, it might be worthwhile to manually use the method `Sum.euler_maclaurin`.

Special optimizations are used for rational hypergeometric series (where the term is a product of polynomials, powers, factorials, binomial coefficients and the like). `N/evalf` sum series of this type very rapidly to high precision. For example, this Ramanujan formula for π can be summed to 10,000 digits in a fraction of a second with a simple command:

```
>>> f = factorial
>>> n = Symbol('n', integer=True)
>>> R = 9801/sqrt(8)/Sum(f(4*n)*((1103+26390*n)/f(n)**4/396**(4*n),
...                      (n, 0, oo))
>>> N(R, 10000)
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
99862803482534211706798214808651328230664709384460955058223172535940812848111745
02841027019385211055596446229489549303819644288109756659334461284756482337867831
...
```

5.6.5 Numerical simplification

The function `nsimplify` attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by `nsimplify` is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, `nsimplify` can be passed a list of constants to include (e.g. π) and a minimum numerical tolerance. Here are some elementary examples:

```
>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355
...
113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
```

(continues on next page)

(continued from previous page)

```
>>> nsimplify(2.0**(1/3.), tolerance=0.001)
635
---
```

$$504$$

```
>>> nsimplify(2.0**(1/3.), tolerance=0.001, full=True)
3
--
\sqrt{2}
```

Here are several more advanced examples:

```
>>> nsimplify(Float('0.130198866629986772369127970337',30), [pi, E])
1
-----
5*pi
---- + 2*E
7
>>> nsimplify(cos(atan('1/3')))
```

$$3\sqrt[3]{10}$$

```
-----
10
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49 8*I
-- + ---
17 17
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
```

$$\frac{1}{2} \frac{I \sqrt{\frac{1}{10} + \frac{\sqrt{5}}{4}}}{\sqrt{\frac{1}{10} + \frac{\sqrt{5}}{4}}}$$

```
>>> nsimplify(I**I, [pi])
-pi
----
2
e
>>> n = Symbol('n')
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
2
pi
---
```

$$6$$

```
>>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])
\sqrt{2} * pi
```

5.7 Structural Details of Code Generation with SymPy

Several submodules in SymPy allow one to generate directly compilable and executable code in a variety of different programming languages from SymPy expressions. In addition, there are functions that generate Python importable objects that can evaluate SymPy expressions very efficiently.

We will start with a brief introduction to the components that make up the code generation capabilities of SymPy.

5.7.1 Introduction

There are four main levels of abstractions:

```
expression
|
code printers
|
code generators
|
autowrap
```

`sympy.utilities.autowrap` (page 1391) uses `codegen`, and `codegen` uses the code printers. `sympy.utilities.autowrap` (page 1391) does everything: it lets you go from SymPy expression to numerical function in the same Python process in one step. `Codegen` is actual code generation, i.e., to compile and use later, or to include in some larger project.

The code printers translate the SymPy objects into actual code, like `abs(x) -> fabs(x)` (for C).

The code printers don't print optimal code in many cases. An example of this is powers in C. `x**2` prints as `pow(x, 2)` instead of `x*x`. Other optimizations (like mathematical simplifications) should happen before the code printers.

Currently, `sympy.simplify.cse_main.cse()` (page 1174) is not applied automatically anywhere in this chain. It ideally happens at the `codegen` level, or somewhere above it.

We will iterate through the levels below.

The following three lines will be used to setup each example:

```
>>> from sympy import *
>>> init_printing(use_unicode=True)
>>> from sympy.abc import a, e, k, n, r, t, x, y, z, T, Z
>>> from sympy.abc import beta, omega, tau
>>> f, g = symbols('f, g', cls=Function)
```

5.7.2 Code printers (sympy.printing)

This is where the meat of code generation is; the translation of SymPy actually more like a lightweight version of `codegen` for Python, and Python (`sympy.printing.pycode.pycode()` (page 1049)), and `sympy.printing.lambdarepr.lambdarepr()` (page 1045), which supports many libraries (like NumPy), and `theano` (`sympy.printing.theanocode.theano_function()` (page 1044)). The code printers are special cases of the other prints in SymPy (str printer, pretty printer, etc.).

An important distinction is that the code printer has to deal with assignments (using the `sympy.codegen.ast.Assignment` (page 395) object). This serves as building blocks for the code printers and hence the `codegen` module. An example that shows the use of `Assignment`:

```
>>> from sympy.codegen.ast import Assignment
>>> mat = Matrix([x, y, z]).T
>>> known_mat = MatrixSymbol('K', 1, 3)
```

(continues on next page)

(continued from previous page)

```
>>> Assignment(known_mat, mat)
K := [x y z]
>>> Assignment(known_mat, mat).lhs
K
>>> Assignment(known_mat, mat).rhs
[x y z]
```

Here is a simple example of printing a C version of a SymPy expression:

```
>>> expr = (Rational(-1, 2) * Z * k * (e**2) / r)
>>> expr
      2
    -Z·e ·k
    -----
      2·r
>>> ccode(expr)
-1.0/2.0*Z*pow(e, 2)*k/r
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, assign_to="E", type_aliases={real: float80})
E = -1.0L/2.0L*Z*powl(e, 2)*k/r;
```

To generate code with some math functions provided by e.g. the C99 standard we need to import functions from `sympy.codegen.cfunctions` (page 409):

```
>>> from sympy.codegen.cfunctions import expm1
>>> ccode(expm1(x), standard='C99')
expm1(x)
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything. A use case for Piecewise:

```
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
    if (x > 0) then
        tau = x + 1
    else
        tau = x
    end if
```

The various printers also tend to support Indexed objects well. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> len_y = 5
>>> mat_1 = IndexedBase('mat_1', shape=(len_y,))
>>> mat_2 = IndexedBase('mat_2', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> eq = Eq(Dy[i], (mat_1[i+1] - mat_1[i]) / (mat_2[i+1] - mat_2[i]))
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Dy[i] = (mat_1[i + 1] - mat_1[i]) / (mat_2[i + 1] - mat_2[i]);
>>> Res = IndexedBase('Res', shape=(len_y,))
>>> j = Idx('j', len_y)
```

(continues on next page)

(continued from previous page)

```

>>> eq = Eq(Res[j], mat_1[j]*mat_2[j])
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=True))
for (var j=0; j<5; j++){
    Res[j] = 0;
}
for (var j=0; j<5; j++){
    for (var j=0; j<5; j++){
        Res[j] = Res[j] + mat_1[j]*mat_2[j];
    }
}
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Res[j] = mat_1[j]*mat_2[j];

```

Custom printing can be defined for certain types by passing a dictionary of “type”: “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument_test, cfunction_string)]. This can be used to call a custom Octave function:

```

>>> custom_functions = {
...     "f": "existing_octave_fcn",
...     "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...           (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])

```

An example of Mathematica code printer:

```

>>> x_ = Function('x')
>>> expr = x_(n*T) * sin((t - n*T) / T)
>>> expr = expr / ((-T*n + t) / T)
>>> expr

```

$$\frac{T \cdot x(T \cdot n) \cdot \sin\left(\frac{-T \cdot n + t}{T}\right)}{-T \cdot n + t}$$

```

>>> expr = summation(expr, (n, -1, 1))
>>> mathematica_code(expr)
T*x[-T]*Sin[(T + t)/T]/(T + t) + T*x[T]*Sin[(-T + t)/T]/(-T + t) + T*x[0]*Sin[t/T]/t

```

We can go through a common expression in different languages we support and see how it works:

```

>>> k, g1, g2, r, I, S = symbols("k, gamma_1, gamma_2, r, I, S")
>>> expr = k * g1 * g2 / (r**3)
>>> expr = expr * 2 * I * S * (3 * (cos(beta))**2 - 1) / 2
>>> expr

```

$$\frac{I \cdot S \cdot \gamma_1 \cdot \gamma_2 \cdot k \cdot (3 \cdot \cos^2(\beta) - 1)}{r^3}$$

```

>>> print(jscode(expr, assign_to="H_is"))

```

(continues on next page)

(continued from previous page)

```

H_is = I*S*gamma_1*gamma_2*k*(3*Math.pow(Math.cos(beta), 2) - 1)/Math.pow(r, 3);
>>> print(ccode(expr, assign_to="H_is", standard='C89'))
H_is = I*S*gamma_1*gamma_2*k*(3*pow(cos(beta), 2) - 1)/pow(r, 3);
>>> print(fcode(expr, assign_to="H_is"))
H_is = I*S*gamma_1*gamma_2*k*(3*cos(beta)**2 - 1)/r**3
>>> print(julia_code(expr, assign_to="H_is"))
H_is = I.*S.*gamma_1.*gamma_2.*k.*(3*cos(beta).^2 - 1)./r.^3
>>> print(octave_code(expr, assign_to="H_is"))
H_is = I.*S.*gamma_1.*gamma_2.*k.*(3*cos(beta).^2 - 1)./r.^3;
>>> print(rust_code(expr, assign_to="H_is"))
H_is = I*S*gamma_1*gamma_2*k*(3*beta.cos().powi(2) - 1)/r.powi(3);
>>> print(mathematica_code(expr))
I*S*gamma_1*gamma_2*k*(3*Cos[beta]^2 - 1)/r^3

```

5.7.3 Codegen (sympy.utilities.codegen)

This module deals with creating compilable code from SymPy expressions. This is lower level than autowrap, as it doesn't actually attempt to compile the code, but higher level than the printers, as it generates compilable files (including header files), rather than just code snippets.

The user friendly functions, here, are `codegen` and `make_routine`. `codegen` takes a list of (variable, expression) pairs and a language (C, F95, and Octave/Matlab are supported). It returns, as strings, a code file and a header file (for relevant languages). The variables are created as functions that return the value of the expression as output.

Note: The `codegen` callable is not in the `sympy` namespace automatically, to use it you must first import `codegen` from `sympy.utilities.codegen`

For instance:

```

>>> from sympy.utilities.codegen import codegen
>>> length, breadth, height = symbols('length, breadth, height')
>>> [(c_name, c_code), (h_name, c_header)] = \
...   codegen(('volume', length*breadth*height), "C99", "test",
...           header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double volume(double breadth, double height, double length) {
    double volume_result;
    volume_result = breadth*height*length;
    return volume_result;
}
>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
double volume(double breadth, double height, double length);
#endif

```

Various flags to codegen let you modify things. The project name for preprocessor instructions can be varied using `project`. Variables listed as global variables in arg `global_vars` will not show up as function arguments.

`language` is a case-insensitive string that indicates the source code language. Currently, C, F95 and Octave are supported. Octave generates code compatible with both Octave and Matlab.

`header` when True, a header is written on top of each source file. `empty` when True, empty lines are used to structure the code. With argument `sequence` a sequence of arguments for the routine can be defined in a preferred order.

`prefix` defines a prefix for the names of the files that contain the source code. If omitted, the name of the first name_expr tuple is used.

`to_files` when True, the code will be written to one or more files with the given prefix.

Here is an example:

```
>>> [(f_name, f_code), header] = codegen(("volume", length*breadth*height),
...   "F95", header=False, empty=False, argument_sequence=(breadth, length),
...   global_vars=(height,))
>>> print(f_code)
REAL*8 function volume(breadth, length)
implicit none
REAL*8, intent(in) :: breadth
REAL*8, intent(in) :: length
volume = breadth*height*length
end function
```

The method `make_routine` creates a Routine object, which represents an evaluation routine for a set of expressions. This is only good for internal use by the CodeGen objects, as an intermediate representation from SymPy expression to generated code. It is not recommended to make a Routine object yourself. You should instead use `make_routine` method. `make_routine` in turn calls the `routine` method of the CodeGen object depending upon the language of choice. This creates the internal objects representing assignments and so on, and creates the Routine class with them.

The various codegen objects such as Routine and Variable aren't SymPy objects (they don't subclass from Basic).

For example:

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, y, x, 6)
>>> routine = make_routine('my_routine', expr)
>>> [arg.result_var for arg in routine.results]
[result5142341681397719428]
>>> [arg.expr for arg in routine.results]
[

$$4 \cdot \sqrt{6} \cdot (4 \cdot x) \cdot \sqrt{\frac{(-y + 2)!}{(y + 3)!}} \cdot e^{-2 \cdot x} \cdot \text{assoc\_laguerre}(-y + 2, 2 \cdot y + 1, 4 \cdot x)$$

]
>>> [arg.name for arg in routine.arguments]
[x, y]
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output:

```
>>> routine = make_routine('fcn', [x*y, Eq(a, 1), Eq(r, x + r), Matrix([[x, 2]])])
>>> [arg.result_var for arg in routine.results]
[result_5397460570204848505]
>>> [arg.expr for arg in routine.results]
[x*y]
>>> [arg.name for arg in routine.arguments]
[x, y, a, r, out_8598435338387848786]
```

We can examine the various arguments more closely:

```
>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
...                                     InOutArgument)
>>> [a.name for a in routine.arguments if isinstance(a, InputArgument)]
[x, y]

>>> [a.name for a in routine.arguments if isinstance(a, OutputArgument)]
[a, out_8598435338387848786]
>>> [a.expr for a in routine.arguments if isinstance(a, OutputArgument)]
[1, [x 2]]

>>> [a.name for a in routine.arguments if isinstance(a, InOutArgument)]
[r]
>>> [a.expr for a in routine.arguments if isinstance(a, InOutArgument)]
[r + x]
```

The full API reference can be viewed [here](#) (page 1396).

5.7.4 Autowrap

Autowrap automatically generates code, writes it to disk, compiles it, and imports it into the current session. Main functions of this module are `autowrap`, `binary_function`, and `ufuncify`.

It also automatically converts expressions containing Indexed objects into summations. The classes `IndexedBase`, `Indexed` and `Idx` represent a matrix element $M[i, j]$. See [Tensor Module](#) (page 1360) for more on this.

`autowrap` creates a wrapper using `f2py` or `Cython` and creates a numerical function.

Note: The `autowrap` callable is not in the `sympy` namespace automatically, to use it you must first import `autowrap` from `sympy.utilities.autowrap`

The callable returned from `autowrap()` is a binary Python function, not a SymPy object. For example:

```
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

The various flags available with `autowrap()` help to modify the services provided by the method. The argument `tempdir` tells `autowrap` to compile the code in a specific directory,

and leave the files intact when finished. For instance:

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy.physics.qho_1d import psi_n
>>> x_ = IndexedBase('x')
>>> y_ = IndexedBase('y')
>>> m = symbols('m', integer=True)
>>> i = Idx('i', m)
>>> qho = autowrap(Eq(y_[i], psi_n(0, x_[i], m, omega)), tempdir='/tmp')
```

Checking the Fortran source code in the directory specified reveals this:

```
subroutine autofunc(m, omega, x, y)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(in), dimension(1:m) :: x
REAL*8, intent(out), dimension(1:m) :: y
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0
do i = 1, m
    y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i &
        )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do

end subroutine
```

Using the argument args along with it changes argument sequence:

```
>>> eq = Eq(y_[i], psi_n(0, x_[i], m, omega))
>>> qho = autowrap(eq, tempdir='/tmp', args=[y, x, m, omega])
```

yields:

```
subroutine autofunc(y, x, m, omega)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(out), dimension(1:m) :: y
REAL*8, intent(in), dimension(1:m) :: x
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0
do i = 1, m
    y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i &
        )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do

end subroutine
```

The argument verbose is boolean, optional and if True, autowrap will not mute the command line backends. This can be helpful for debugging.

The argument language and backend are used to change defaults: Fortran and f2py to C and Cython. The argument helpers is used to define auxiliary expressions needed for the main expression. If the main expression needs to call a specialized function it should be put in the

helpers iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (`<function_name>`, `<sympy_expression>`, `<arguments>`). It is mandatory to supply an argument sequence to helper routines.

Another method available at the autowrap level is `binary_function`. It returns a sympy function. The advantage is that we can have very fast functions as compared to SymPy speeds. This is because we will be using compiled functions with Sympy attributes and methods. An illustration:

```
>>> from sympy.utilities.autowrap import binary_function
>>> from sympy.physics.hydrogen import R_nl
>>> psi_nl = R_nl(1, 0, a, r)
>>> f = binary_function('f', psi_nl)
>>> f(a, r).evalf(3, subs={a: 1, r: 2})
0.766
```

While NumPy operations are very efficient for vectorized data but they sometimes incur unnecessary costs when chained together. Consider the following operation

```
>>> x = get_numpy_array(...)
>>> y = sin(x) / x
```

The operators `sin` and `/` call routines that execute tight for loops in C. The resulting computation looks something like this

```
for(int i = 0; i < n; i++)
{
    temp[i] = sin(x[i]);
}
for(int i = i; i < n; i++)
{
    y[i] = temp[i] / x[i];
}
```

This is slightly sub-optimal because

1. We allocate an extra `temp` array
2. We walk over `x` memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

```
for(int i = i; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}
```

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, SymPy is able to generate efficient low-level C or Fortran code. It can then depend on projects like Cython or f2py to compile and reconnect that code back up to Python. Fortunately this process is well automated and a SymPy user wishing to make use of this code generation should call the `ufuncify` function.

`ufuncify` is the third method available with Autowrap module. It basically implies 'Universal functions' and follows an ideology set by NumPy. The main point of `ufuncify` as compared to autowrap is that it allows arrays as arguments and can operate in an element-by-element fashion. The core operation done element-wise is in accordance to Numpy's array broadcasting rules. See [this](#) for more.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```
>>> from sympy.utilities.autowrap import ufuncify
>>> f = ufuncify([x], expr)
```

This function `f` consumes and returns a NumPy array. Generally `ufuncify` performs at least as well as `lambdify`. If the expression is complicated then `ufuncify` often significantly outperforms the NumPy backed solution. Jensen has a good [blog post](#) on this topic.

Let us see an example for some quantitative analysis:

```
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, 1, x, 6)
>>> expr
      -2·x
8·x·(-4·x + 4)·e
—————
      3
```

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. By default `lambdify` relies on implementations in the math standard library. Naturally, Raw Python is faster than SymPy. However it also supports `mpmath` and most notably, `numpy`. Using the NumPy library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

Let us compare the speeds:

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.utilities.lambdify import lambdify
>>> fn_numpy = lambdify(x, expr, 'numpy')
>>> fn_fortran = ufuncify([x], expr, backend='f2py')
>>> from numpy import linspace
>>> xx = linspace(0, 1, 5)
>>> fn_numpy(xx)
[ 0.          1.21306132  0.98101184  0.44626032  0.          ]
>>> fn_fortran(xx)
[ 0.          1.21306132  0.98101184  0.44626032  0.          ]
>>> import timeit
>>> timeit.timeit('fn_numpy(xx)', 'from __main__ import fn_numpy, xx', number=10000)
0.18891601900395472
>>> timeit.timeit('fn_fortran(xx)', 'from __main__ import fn_fortran, xx',
↳ number=10000)
0.004707066000264604
```

The options available with `ufuncify` are more or less the same as those available with `autowrap`.

There are other facilities available with SymPy to do efficient numeric computation. See [this](#) (page 423) page for a comparison among them.

5.7.5 Classes and functions for rewriting expressions (sympy.codegen.rewriting)

Classes and functions useful for rewriting expressions for optimized code generation. Some languages (or standards thereof), e.g. C99, offer specialized math functions for better per-

formance and/or precision.

Using the `optimize` function in this module, together with a collection of rules (represented as instances of `Optimization`), one can rewrite the expressions for this purpose:

```
>>> from sympy import Symbol, exp, log
>>> from sympy.codegen.rewriting import optimize, optims_c99
>>> x = Symbol('x')
>>> optimize(3*exp(2*x) - 3, optims_c99)
3*expm1(2*x)
>>> optimize(exp(2*x) - 3, optims_c99)
exp(2*x) - 3
>>> optimize(log(3*x + 3), optims_c99)
loglp(x) + log(3)
>>> optimize(log(2*x + 3), optims_c99)
log(2*x + 3)
```

The `optims_c99` imported above is tuple containing the following instances (which may be imported from `sympy.codegen.rewriting`):

- `expm1_opt`
- `loglp_opt`
- `exp2_opt`
- `log2_opt`
- `log2const_opt`

class `sympy.codegen.rewriting.Optimization`(`cost_function=None`, `priority=1`)
Abstract base class for rewriting optimization.

Subclasses should implement `__call__` taking an expression as argument.

Parameters `cost_function` : callable returning number

priority : number

class `sympy.codegen.rewriting.ReplaceOptim`(`query`, `value`, `**kwargs`)
Rewriting optimization calling `replace` on expressions.

The instance can be used as a function on expressions for which it will apply the `replace` method (see `sympy.core.basic.Basic.replace()` (page 104)).

Parameters `query` : first argument passed to `replace`

value : second argument passed to `replace`

Examples

```
>>> from sympy import Symbol, Pow
>>> from sympy.codegen.rewriting import ReplaceOptim
>>> from sympy.codegen.cfunctions import exp2
>>> x = Symbol('x')
>>> exp2_opt = ReplaceOptim(lambda p: p.is_Pow and p.base == 2,
... lambda p: exp2(p.exp))
>>> exp2_opt(2**x)
exp2(x)
```

`sympy.codegen.rewriting.create_expand_pow_optimization`(`limit`)
Creates an instance of `ReplaceOptim` (page 390) for expanding `Pow`.

The requirements for expansions are that the base needs to be a symbol and the exponent needs to be an integer (and be less than or equal to `limit`).

Parameters `limit` : int

The highest power which is expanded into multiplication.

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.codegen.rewriting import create_expand_pow_optimization
>>> x = Symbol('x')
>>> expand_opt = create_expand_pow_optimization(3)
>>> expand_opt(x**5 + x**3)
x**5 + x*x*x
>>> expand_opt(x**5 + x**3 + sin(x)**3)
x**5 + x*x*x + sin(x)**3
```

`sympy.codegen.rewriting.optimize(expr, optimizations)`

Apply optimizations to an expression.

Parameters `expr` : expression

optimizations : iterable of Optimization instances

The optimizations will be sorted with respect to priority (highest first).

Examples

```
>>> from sympy import log, Symbol
>>> from sympy.codegen.rewriting import optims_c99, optimize
>>> x = Symbol('x')
>>> optimize(log(x+3)/log(2) + log(x**2 + 1), optims_c99)
log1p(x**2) + log2(x + 3)
```

5.7.6 Tools for simplifying expressions using approximations (sympy.codegen.approximations)

class `sympy.codegen.approximations.SeriesApprox(bounds, reltol, max_order=4, n_point_checks=4, **kwargs)`

Approximates functions by expanding them as a series

Parameters `bounds` : dict

Mapping expressions to length 2 tuple of bounds (low, high).

reltol : number

Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

max_order : int

Largest order to include in series expansion

n_point_checks : int (even)

The validity of an expansion (with respect to `reltol`) is checked at discrete points (linearly spaced over the bounds of the variable). The number of points used in this numerical check is given by this number.

Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x, y
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SeriesApprox
>>> bounds = {x: (-.1, .1), y: (pi-1, pi+1)}
>>> series_approx2 = SeriesApprox(bounds, reltol=1e-2)
>>> series_approx3 = SeriesApprox(bounds, reltol=1e-3)
>>> series_approx8 = SeriesApprox(bounds, reltol=1e-8)
>>> expr = sin(x)*sin(y)
>>> optimize(expr, [series_approx2])
x*(-y + (y - pi)**3/6 + pi)
>>> optimize(expr, [series_approx3])
(-x**3/6 + x)*sin(y)
>>> optimize(expr, [series_approx8])
sin(x)*sin(y)
```

class `sympy.codegen.approximations.SumApprox`(`bounds`, `reltol`, `**kwargs`)

Approximates sum by neglecting small terms

If terms are expressions which can be determined to be monotonic, then bounds for those expressions are added.

Parameters `bounds` : dict

Mapping expressions to length 2 tuple of bounds (low, high).

reltol : number

Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

Examples

```
>>> from sympy import exp
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SumApprox
>>> bounds = {x: (-1, 1), y: (1000, 2000), z: (-10, 3)}
>>> sum_approx3 = SumApprox(bounds, reltol=1e-3)
>>> sum_approx2 = SumApprox(bounds, reltol=1e-2)
>>> sum_approx1 = SumApprox(bounds, reltol=1e-1)
>>> expr = 3*(x + y + exp(z))
>>> optimize(expr, [sum_approx3])
3*(x + y + exp(z))
>>> optimize(expr, [sum_approx2])
3*y + 3*exp(z)
>>> optimize(expr, [sum_approx1])
3*y
```

5.7.7 Classes for abstract syntax trees (sympy.codegen.ast)

Types used to represent a full function/module as an Abstract Syntax Tree.

Most types are small, and are merely used as tokens in the AST. A tree diagram has been included below to illustrate the relationships between the AST types.

AST Type Tree

```
*Basic*
|--->Assignment
|   |--->AugmentedAssignment
|       |--->AddAugmentedAssignment
|       |--->SubAugmentedAssignment
|       |--->MulAugmentedAssignment
|       |--->DivAugmentedAssignment
|       |--->ModAugmentedAssignment
|
|--->CodeBlock
|
|--->Token
|   |--->Attribute
|   |--->For
|   |--->String
|       |--->QuotedString
|       |--->Comment
|   |--->Type
|       |--->IntBaseType
|           |--->_SizedIntType
|               |--->SignedIntType
|               |--->UnsignedIntType
|       |--->FloatBaseType
|           |--->FloatType
|           |--->ComplexBaseType
|               |--->ComplexType
|   |--->Node
|       |--->Variable
|       |   |--->Pointer
|       |--->FunctionPrototype
|           |--->FunctionDefinition
|   |--->Element
|   |--->Declaration
|   |--->While
|   |--->Scope
|   |--->Stream
|   |--->Print
|   |--->FunctionCall
|   |--->BreakToken
|   |--->ContinueToken
|   |--->NoneToken
|
|--->Statement
|--->Return
```

Predefined types

A number of Type instances are provided in the `sympy.codegen.ast` module for convenience. Perhaps the two most common ones for code-generation (of numeric codes) are `float32` and `float64` (known as single and double precision respectively). There are also precision generic versions of Types (for which the codeprinters selects the underlying data type at time of printing): `real`, `integer`, `complex_`, `bool_`.

The other Type instances defined are:

- `intc`: Integer type used by C's "int".
- `intp`: Integer type used by C's "unsigned".
- `int8`, `int16`, `int32`, `int64`: n-bit integers.
- `uint8`, `uint16`, `uint32`, `uint64`: n-bit unsigned integers.
- `float80`: known as "extended precision" on modern x86/amd64 hardware.
- `complex64`: Complex number represented by two `float32` numbers
- `complex128`: Complex number represented by two `float64` numbers

Using the nodes

It is possible to construct simple algorithms using the AST nodes. Let's construct a loop applying Newton's method:

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import While, Assignment, aug_assign, Print
>>> t, dx, x = symbols('tol delta val')
>>> expr = cos(x) - x**3
>>> whl = While(abs(dx) > t, [
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx),
...     Print([x])
... ])
>>> from sympy.printing import pycode
>>> py_str = pycode(whl)
>>> print(py_str)
while (abs(delta) > tol):
    delta = (val**3 - math.cos(val))/(-3*val**2 - math.sin(val))
    val += delta
    print(val)
>>> import math
>>> tol, val, delta = 1e-5, 0.5, float('inf')
>>> exec(py_str)
1.1121416371
0.909672693737
0.867263818209
0.865477135298
0.865474033111
>>> print('%3.1g' % (math.cos(val) - val**3))
-3e-11
```

If we want to generate Fortran code for the same while loop we simple call `fcode`:


```
>>> from sympy.printing.fcode import fcode
>>> print(fcode(whl, standard=2003, source_format='free'))
do while (abs(delta) > tol)
    delta = (val**3 - cos(val))/(-3*val**2 - sin(val))
    val = val + delta
    print *, val
end do
```

There is a function constructing a loop (or a complete function) like this in `sympy.codegen.algorithms` (page 421).

class `sympy.codegen.ast.Assignment`

Represents variable assignment for code generation.

Parameters `lhs` : Expr

Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include `Symbol`, `MatrixSymbol`, `MatrixElement`, and `Indexed`. Types that subclass these types are also supported.

rhs : Expr

Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a `Matrix` type can be assigned to `MatrixSymbol`, but not to `Symbol`, as the dimensions will not align.

Examples

```
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> from sympy.codegen.ast import Assignment
>>> x, y, z = symbols('x, y, z')
>>> Assignment(x, y)
Assignment(x, y)
>>> Assignment(x, 0)
Assignment(x, 0)
>>> A = MatrixSymbol('A', 1, 3)
>>> mat = Matrix([x, y, z]).T
>>> Assignment(A, mat)
Assignment(A, Matrix([[x, y, z]]))
>>> Assignment(A[0, 1], x)
Assignment(A[0, 1], x)
```

class `sympy.codegen.ast.Attribute` (possibly parametrized)

For use with `sympy.codegen.ast.Node` (page 402) (which takes instances of `Attribute` as `attrs`).

Parameters `name` : str

parameters : Tuple

Examples

```
>>> from sympy.codegen.ast import Attribute
>>> volatile = Attribute('volatile')
>>> volatile
volatile
>>> print(repr(volatile))
Attribute(String('volatile'))
>>> a = Attribute('foo', [1, 2, 3])
>>> a
foo(1, 2, 3)
>>> a.parameters == (1, 2, 3)
True
```

class sympy.codegen.ast.**AugmentedAssignment**

Base class for augmented assignments

class sympy.codegen.ast.**BreakToken**

Represents 'break' in C/Python ('exit' in Fortran).

Use the premade instance `break_` or instantiate manually.

Examples

```
>>> from sympy.printing import ccode, fcode
>>> from sympy.codegen.ast import break_
>>> ccode(break_)
'break'
>>> fcode(break_, source_format='free')
'exit'
```

class sympy.codegen.ast.**CodeBlock**

Represents a block of code

For now only assignments are supported. This restriction will be lifted in the future.

Useful attributes on this object are:

left_hand_sides: Tuple of left-hand sides of assignments, in order.

right_hand_sides: Tuple of right-hand sides of assignments, in order.

free_symbols: Free symbols of the expressions in the right-hand sides which do not appear in the left-hand side of an assignment.

Useful methods on this object are:

topological_sort: Class method. Return a CodeBlock with assignments sorted so that variables are assigned before they are used.

cse: Return a new CodeBlock with common subexpressions eliminated and pulled out as assignments.

Example

```
>>> from sympy import symbols, ccode
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y = symbols('x y')
>>> c = CodeBlock(Assignment(x, 1), Assignment(y, x + 1))
```

(continues on next page)

(continued from previous page)

```
>>> print(ccode(c))
x = 1;
y = x + 1;
```

cse(symbols=None, optimizations=None, postprocess=None, order='canonical')

Return a new code block with common subexpressions eliminated

See the docstring of `sympy.simplify.cse_main.cse()` (page 1174) for more information.

Examples

```
>>> from sympy import symbols, sin
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```
>>> c = CodeBlock(
...     Assignment(x, 1),
...     Assignment(y, sin(x) + 1),
...     Assignment(z, sin(x) - 1),
... )
...
>>> c.cse()
CodeBlock(
    Assignment(x, 1),
    Assignment(x0, sin(x)),
    Assignment(y, x0 + 1),
    Assignment(z, x0 - 1)
)
```

classmethod topological_sort(assignments)

Return a CodeBlock with topologically sorted assignments so that variables are assigned before they are used.

The existing order of assignments is preserved as much as possible.

This function assumes that variables are assigned to only once.

This is a class constructor so that the default constructor for CodeBlock can error when variables are used before they are assigned.

Example

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```
>>> assignments = [
...     Assignment(x, y + z),
...     Assignment(y, z + 1),
...     Assignment(z, 2),
... ]
>>> CodeBlock.topological_sort(assignments)
```

(continues on next page)

(continued from previous page)

```
CodeBlock(  
    Assignment(z, 2),  
    Assignment(y, z + 1),  
    Assignment(x, y + z)  
)
```

class sympy.codegen.ast.**Comment**

Represents a comment.

class sympy.codegen.ast.**ComplexType**

Represents a complex floating point number.

class sympy.codegen.ast.**ContinueToken**

Represents 'continue' in C/Python ('cycle' in Fortran)

Use the premade instance `continue_` or instantiate manually.

Examples

```
>>> from sympy.printing import ccode, fcode  
>>> from sympy.codegen.ast import continue_  
>>> ccode(continue_)  
'continue'  
>>> fcode(continue_, source_format='free')  
'cycle'
```

class sympy.codegen.ast.**Declaration**

Represents a variable declaration

Parameters `variable` : Variable

Examples

```
>>> from sympy import Symbol  
>>> from sympy.codegen.ast import Declaration, Type, Variable, integer, untyped  
>>> z = Declaration('z')  
>>> z.variable.type == untyped  
True  
>>> z.variable.value == None  
True
```

class sympy.codegen.ast.**Element**

Element in (a possibly N-dimensional) array.

Examples

```
>>> from sympy.codegen.ast import Element  
>>> elem = Element('x', 'ijk')  
>>> elem.symbol.name == 'x'  
True  
>>> elem.indices  
(i, j, k)
```

(continues on next page)

(continued from previous page)

```
>>> from sympy import ccode
>>> ccode(elem)
'x[i][j][k]'
>>> ccode(Element('x', 'ijk', strides='lmn', offset='o'))
'x[i*l + j*m + k*n + o]'
```

class sympy.codegen.ast.FloatBaseType

Represents a floating point number type.

cast_nocheckalias of `sympy.core.numbers.Float` (page 138)**class** sympy.codegen.ast.FloatType

Represents a floating point type with fixed bit width.

Base 2 & one sign bit is assumed.

Parameters **name** : str

Name of the type.

nbits : integer

Number of bits used (storage).

nmant : integer

Number of bits used to represent the mantissa.

nexp : integer

Number of bits used to represent the mantissa.

Examples

```
>>> from sympy import S, Float
>>> from sympy.codegen.ast import FloatType
>>> half_precision = FloatType('f16', nbits=16, nmant=10, nexp=5)
>>> half_precision.max
65504
>>> half_precision.tiny == S(2)**-14
True
>>> half_precision.eps == S(2)**-10
True
>>> half_precision.dig == 3
True
>>> half_precision.decimal_dig == 5
True
>>> half_precision.cast_check(1.0)
1.0
>>> half_precision.cast_check(1e5)
Traceback (most recent call last):
...
ValueError: Maximum value for data type smaller than new value.
```

cast_nocheck(value)

Casts without checking if out of bounds or subnormal.

decimal_dig

Number of digits needed to store & load without loss.

Number of decimal digits needed to guarantee that two consecutive conversions (float -> text -> float) to be idempotent. This is useful when one do not want to loose precision due to rounding errors when storing a floating point value as text.

dig

Number of decimal digits that are guaranteed to be preserved in text.

When converting text -> float -> text, you are guaranteed that at least **dig** number of digits are preserved with respect to rounding or overflow.

eps

Difference between 1.0 and the next representable value.

max

Maximum value representable.

max_exponent

The largest positive number n , such that $2^{*(n - 1)}$ is a representable finite value.

min_exponent

The lowest negative number n , such that $2^{*(n - 1)}$ is a valid normalized number.

tiny

The minimum positive normalized value.

class sympy.codegen.ast.For

Represents a 'for-loop' in the code.

Expressions are of the form:

"for target in iter: body..."

Parameters **target** : symbol

iter : iterable body : CodeBlock or iterable

! When passed an iterable it is used to instantiate a CodeBlock.

Examples

```
>>> from sympy import symbols, Range
>>> from sympy.codegen.ast import aug_assign, For
>>> x, i, j, k = symbols('x i j k')
>>> for_i = For(i, Range(10), [aug_assign(x, '+', i*j*k)])
>>> for_i
For(i, iterable=Range(0, 10, 1), body=CodeBlock(
  AddAugmentedAssignment(x, i*j*k)
))
>>> for_ji = For(j, Range(7), [for_i])
>>> for_ji
For(j, iterable=Range(0, 7, 1), body=CodeBlock(
  For(i, iterable=Range(0, 10, 1), body=CodeBlock(
    AddAugmentedAssignment(x, i*j*k)
  ))
))
>>> for_kji = For(k, Range(5), [for_ji])
>>> for_kji
For(k, iterable=Range(0, 5, 1), body=CodeBlock(
  For(j, iterable=Range(0, 7, 1), body=CodeBlock(
    For(i, iterable=Range(0, 10, 1), body=CodeBlock(
```

(continues on next page)

(continued from previous page)

```

        AddAugmentedAssignment(x, i*j*k)
    ))
))
))

```

class `sympy.codegen.ast.FunctionCall`

Represents a call to a function in the code.

Parameters `name` : str
function_args : Tuple

Examples

```

>>> from sympy.codegen.ast import FunctionCall
>>> from sympy.printing.pycode import pycode
>>> fcall = FunctionCall('foo', 'bar baz'.split())
>>> print(pycode(fcall))
foo(bar, baz)

```

class `sympy.codegen.ast.FunctionDefinition`

Represents a function definition in the code.

Parameters `return_type` : Type
name : str
parameters: iterable of Variable instances
body : CodeBlock or iterable
attrs : iterable of Attribute instances

Examples

```

>>> from sympy import symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> from sympy.printing.ccode import ccode
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
>>> from sympy.codegen.ast import FunctionDefinition, Return
>>> body = [Return(x*y)]
>>> fd = FunctionDefinition.from_FunctionPrototype(fp, body)
>>> print(ccode(fd))
double foo(double x, double y){
    return x*y;
}

```

class `sympy.codegen.ast.FunctionPrototype`

Represents a function prototype

Allows the user to generate forward declaration in e.g. C/C++.

Parameters `return_type` : Type
name : str

parameters: iterable of Variable instances

attrs : iterable of Attribute instances

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> from sympy.printing.ccode import ccode
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
```

class sympy.codegen.ast.IntBaseType

Integer base type, contains no size information.

class sympy.codegen.ast.Node

Subclass of Token, carrying the attribute 'attrs' (Tuple)

Examples

```
>>> from sympy.codegen.ast import Node, value_const, pointer_const
>>> n1 = Node([value_const])
>>> n1.attr_params('value_const') # get the parameters of attribute (by name)
()
>>> from sympy.codegen.fnodes import dimension
>>> n2 = Node([value_const, dimension(5, 3)])
>>> n2.attr_params(value_const) # get the parameters of attribute (by Attribute_
↳instance)
()
>>> n2.attr_params('dimension') # get the parameters of attribute (by name)
(5, 3)
>>> n2.attr_params(pointer_const) is None
True
```

attr_params(looking_for)

Returns the parameters of the Attribute with name looking_for in self.attrs

class sympy.codegen.ast.NoneToken

The AST equivalence of Python's NoneType

The corresponding instance of Python's None is none.

Examples

```
>>> from sympy.codegen.ast import none, Variable
>>> from sympy.printing.pycode import pycode
>>> print(pycode(Variable('x').as_Declaration(value=none)))
x = None
```

class sympy.codegen.ast.Pointer

Represents a pointer. See Variable.

Examples

Can create instances of Element:

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Pointer
>>> i = Symbol('i', integer=True)
>>> p = Pointer('x')
>>> p[i+1]
Element(x, indices=((i + 1),))
```

class sympy.codegen.ast.Print

Represents print command in the code.

Parameters formatstring : str

***args** : Basic instances (or convertible to such through sympify)

Examples

```
>>> from sympy.codegen.ast import Print
>>> from sympy.printing.pycode import pycode
>>> print(pycode(Print('x y'.split(), "coordinate: %12.5g %12.5g")))
print("coordinate: %12.5g %12.5g" % (x, y))
```

class sympy.codegen.ast.QuotedString

Represents a string which should be printed with quotes.

class sympy.codegen.ast.Return

Represents a return command in the code.

class sympy.codegen.ast.Scope

Represents a scope in the code.

Parameters body : CodeBlock or iterable

When passed an iterable it is used to instantiate a CodeBlock.

class sympy.codegen.ast.SignedIntType

Represents a signed integer type.

class sympy.codegen.ast.Stream

Represents a stream.

There are two predefined Stream instances stdout & stderr.

Parameters name : str

Examples

```
>>> from sympy import Symbol
>>> from sympy.printing.pycode import pycode
>>> from sympy.codegen.ast import Print, stderr, QuotedString
>>> print(pycode(Print(['x'], file=stderr)))
print(x, file=sys.stderr)
>>> x = Symbol('x')
>>> print(pycode(Print([QuotedString('x')], file=stderr))) # print literally "x"
print("x", file=sys.stderr)
```

class sympy.codegen.ast.String

SymPy object representing a string.

Atomic object which is not an expression (as opposed to Symbol).

Parameters text : str

Examples

```
>>> from sympy.codegen.ast import String
>>> f = String('foo')
>>> f
foo
>>> str(f)
'foo'
>>> f.text
'foo'
>>> print(repr(f))
String('foo')
```

class sympy.codegen.ast.Token

Base class for the AST types.

Defining fields are set in `__slots__`. Attributes (defined in `__slots__`) are only allowed to contain instances of Basic (unless atomic, see String). The arguments to `__new__()` correspond to the attributes in the order defined in `__slots__`. The `''defaults` class attribute is a dictionary mapping attribute names to their default values.

Subclasses should not need to override the `__new__()` method. They may define a class or static method named `_construct_<attr>` for each attribute to process the value passed to `__new__()`. Attributes listed in the class attribute `not_in_args` are not passed to `sympy.Basic`.

kwargs (exclude=(), apply=None)

Get instance's attributes as dict of keyword arguments.

Parameters:

exclude [collection of str] Collection of keywords to exclude.

apply [callable, optional] Function to apply to all values.

class sympy.codegen.ast.Type

Represents a type.

The naming is a super-set of NumPy naming. Type has a classmethod `from_expr` which offer type deduction. It also has a method `cast_check` which casts the argument to its type, possibly raising an exception if rounding error is not within tolerances, or if the value is not representable by the underlying data type (e.g. unsigned integers).

Parameters name : str

Name of the type, e.g. object, int16, float16 (where the latter two would use the Type sub-classes IntType and FloatType respectively).

If a Type instance is given, the said instance is returned.

References

[R25] (page 1857)

Examples

```
>>> from sympy.codegen.ast import Type
>>> t = Type.from_expr(42)
>>> t
integer
>>> print(repr(t))
IntBaseType(String('integer'))
>>> from sympy.codegen.ast import uint8
>>> uint8.cast_check(-1)
Traceback (most recent call last):
...
ValueError: Minimum value for data type bigger than new value.
>>> from sympy.codegen.ast import float32
>>> v6 = 0.123456
>>> float32.cast_check(v6)
0.123456
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
...
ValueError: Casting gives a significantly different value.
>>> boost_mp50 = Type('boost::multiprecision::cpp_dec_float_50')
>>> from sympy import Symbol
>>> from sympy.printing.cxxcode import cxxcode
>>> from sympy.codegen.ast import Declaration, Variable
>>> cxxcode(Declaration(Variable('x', type=boost_mp50)))
'boost::multiprecision::cpp_dec_float_50 x'
```

cast_check(value, rtol=None, atol=0, limits=None, precision_targets=None)
Casts a value to the data type of the instance.

Parameters **value** : number

rtol : floating point number

Relative tolerance. (will be deduced if not given).

atol : floating point number

Absolute tolerance (in addition to rtol).

limits : dict

Values given by `limits.h`, x86/IEEE754 defaults if not given. Default: `default_limits`.

type_aliases : dict

Maps substitutions for Type, e.g. {integer: int64, real: float32}

Examples

```
>>> from sympy.codegen.ast import Type, integer, float32, int8
>>> integer.cast_check(3.0) == 3
True
>>> float32.cast_check(1e-40)
Traceback (most recent call last):
...
ValueError: Minimum value for data type bigger than new value.
>>> int8.cast_check(256)
Traceback (most recent call last):
...
ValueError: Maximum value for data type smaller than new value.
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
...
ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float64
>>> float64.cast_check(v10)
12345.67894
>>> from sympy import Float
>>> v18 = Float('0.123456789012345646')
>>> float64.cast_check(v18)
Traceback (most recent call last):
...
ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float80
>>> float80.cast_check(v18)
0.123456789012345649
```

classmethod `from_expr(expr)`

Deduces type from an expression or a Symbol.

Parameters `expr` : number or SymPy object

The type will be deduced from type or properties.

Raises `ValueError` when type deduction fails.

Examples

```
>>> from sympy.codegen.ast import Type, integer, complex_
>>> Type.from_expr(2) == integer
True
>>> from sympy import Symbol
>>> Type.from_expr(Symbol('z', complex=True)) == complex_
True
>>> Type.from_expr(sum)
Traceback (most recent call last):
...
ValueError: Could not deduce type from expr.
```

class `sympy.codegen.ast.UnsignedIntType`

Represents an unsigned integer type.

class `sympy.codegen.ast.Variable`

Represents a variable

Parameters **symbol** : Symbol

type : Type (optional)

Type of the variable.

attrs : iterable of Attribute instances

Will be stored as a Tuple.

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, float32, integer
>>> x = Symbol('x')
>>> v = Variable(x, type=float32)
>>> v.attrs
()
>>> v == Variable('x')
False
>>> v == Variable('x', type=float32)
True
>>> v
Variable(x, type=float32)
```

One may also construct a `Variable` instance with the type deduced from assumptions about the symbol using the deduced classmethod:

```
>>> i = Symbol('i', integer=True)
>>> v = Variable.deduced(i)
>>> v.type == integer
True
>>> v == Variable('i')
False
>>> from sympy.codegen.ast import value_const
>>> value_const in v.attrs
False
>>> w = Variable('w', attrs=[value_const])
>>> w
Variable(w, attrs=(value_const,))
>>> value_const in w.attrs
True
>>> w.as_Declaration(value=42)
Declaration(Variable(w, value=42, attrs=(value_const,)))
```

as_Declaration(**kwargs)

Convenience method for creating a `Declaration` instance.

If the variable of the `Declaration` need to wrap a modified variable keyword arguments may be passed (overriding e.g. the value of the `Variable` instance).

Examples

```
>>> from sympy.codegen.ast import Variable
>>> x = Variable('x')
>>> decl1 = x.as_Declaration()
```

(continues on next page)

(continued from previous page)

```
>>> decl1.variable.value == None
True
>>> decl2 = x.as_Declaration(value=42.0)
>>> decl2.variable.value == 42
True
```

classmethod deduced(symbol, value=None, attrs=(), cast_check=True)

Alt. constructor with type deduction from `Type.from_expr`.

Deduces type primarily from `symbol`, secondarily from `value`.

Parameters `symbol` : Symbol

value : expr

(optional) value of the variable.

attrs : iterable of Attribute instances

cast_check : bool

Whether to apply `Type.cast_check` on `value`.

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, complex_
>>> n = Symbol('n', integer=True)
>>> str(Variable.deduced(n).type)
'integer'
>>> x = Symbol('x', real=True)
>>> v = Variable.deduced(x)
>>> v.type
real
>>> z = Symbol('z', complex=True)
>>> Variable.deduced(z).type == complex_
True
```

class `sympy.codegen.ast.While`

Represents a 'for-loop' in the code.

Expressions are of the form:

"while condition: body..."

Parameters `condition` : expression convertible to Boolean

body : CodeBlock or iterable

When passed an iterable it is used to instantiate a CodeBlock.

Examples

```
>>> from sympy import symbols, Gt, Abs
>>> from sympy.codegen import aug_assign, Assignment, While
>>> x, dx = symbols('x dx')
>>> expr = 1 - x**2
```

(continues on next page)

(continued from previous page)

```
>>> whl = While(Gt(Abs(dx), 1e-9), [
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx)
... ])
```

`sympy.codegen.ast.aug_assign(lhs, op, rhs)`

Create 'lhs op= rhs'.

Represents augmented variable assignment for code generation. This is a convenience function. You can also use the AugmentedAssignment classes directly, like `AddAugmentedAssignment(x, y)`.

Parameters `lhs` : Expr

Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include `Symbol`, `MatrixSymbol`, `MatrixElement`, and `Indexed`. Types that subclass these types are also supported.

op : str

Operator (+, -, /, *, %).

rhs : Expr

Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a `Matrix` type can be assigned to `MatrixSymbol`, but not to `Symbol`, as the dimensions will not align.

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import aug_assign
>>> x, y = symbols('x, y')
>>> aug_assign(x, '+', y)
AddAugmentedAssignment(x, y)
```

5.7.8 Special C math functions (`sympy.codegen.cfunctions`)

This module contains SymPy functions corresponding to special math functions in the C standard library (since C99, also available in C++11).

The functions defined in this module allows the user to express functions such as `expm1` as a SymPy function for symbolic manipulation.

class `sympy.codegen.cfunctions.Cbrt`

Represents the cube root function.

The reason why one would use `Cbrt(x)` over `cbert(x)` is that the latter is internally represented as `Pow(x, Rational(1, 3))` which may not be what one wants when doing code-generation.

See also:

[Sqrt](#) (page 410)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Cbrt
>>> Cbrt(x)
Cbrt(x)
>>> Cbrt(x).diff(x)
1/(3*x**(2/3))
```

fdiff(argindex=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**Sqrt**

Represents the square root function.

The reason why one would use `Sqrt(x)` over `sqrt(x)` is that the latter is internally represented as `Pow(x, S.Half)` which may not be what one wants when doing code-generation.

See also:

[Cbrt](#) (page 409)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Sqrt
>>> Sqrt(x)
Sqrt(x)
>>> Sqrt(x).diff(x)
1/(2*sqrt(x))
```

fdiff(argindex=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**exp2**

Represents the exponential function with base two.

The benefit of using `exp2(x)` over `2**x` is that the latter is not as efficient under finite precision arithmetic.

See also:

[log2](#) (page 412)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import exp2
>>> exp2(2).evalf() == 4
True
>>> exp2(x).diff(x)
log(2)*exp2(x)
```

fdiff(argindex=1)

Returns the first derivative of this function.

class `sympy.codegen.cfunctions.expm1`

Represents the exponential function minus one.

The benefit of using `expm1(x)` over `exp(x) - 1` is that the latter is prone to cancellation under finite precision arithmetic when `x` is close to zero.

See also:

[log1p](#) (page 412)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import expm1
>>> '%.0e' % expm1(1e-99).evalf()
'1e-99'
>>> from math import exp
>>> exp(1e-99) - 1
0.0
>>> expm1(x).diff(x)
exp(x)
```

fdiff(argindex=1)

Returns the first derivative of this function.

class `sympy.codegen.cfunctions.fma`

Represents “fused multiply add”.

The benefit of using `fma(x, y, z)` over `x*y + z` is that, under finite precision arithmetic, the former is supported by special instructions on some CPUs.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.cfunctions import fma
>>> fma(x, y, z).diff(x)
y
```

fdiff(argindex=1)

Returns the first derivative of this function.

class `sympy.codegen.cfunctions.hypot`

Represents the hypotenuse function.

The hypotenuse function is provided by e.g. the `math` library in the C99 standard, hence one may want to represent the function symbolically when doing code-generation.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.codegen.cfunctions import hypot
>>> hypot(3, 4).evalf() == 5
True
>>> hypot(x, y)
hypot(x, y)
```

(continues on next page)

(continued from previous page)

```
>>> hypot(x, y).diff(x)
x/hypot(x, y)
```

fdiff(argindex=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log10**

Represents the logarithm function with base ten.

See also:[log2](#) (page 412)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log10
>>> log10(100).evalf() == 2
True
>>> log10(x).diff(x)
1/(x*log(10))
```

fdiff(argindex=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log1p**

Represents the natural logarithm of a number plus one.

The benefit of using `log1p(x)` over `log(x + 1)` is that the latter is prone to cancellation under finite precision arithmetic when `x` is close to zero.

See also:[expm1](#) (page 410)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log1p
>>> from sympy.core.function import expand_log
>>> '%.0e' % expand_log(log1p(1e-99)).evalf()
'1e-99'
>>> from math import log
>>> log(1 + 1e-99)
0.0
>>> log1p(x).diff(x)
1/(x + 1)
```

fdiff(argindex=1)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log2**

Represents the logarithm function with base two.

The benefit of using `log2(x)` over `log(x)/log(2)` is that the latter is not as efficient under finite precision arithmetic.

See also:

[exp2](#) (page 410), [log10](#) (page 412)

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log2
>>> log2(4).evalf() == 2
True
>>> log2(x).diff(x)
1/(x*log(2))
```

fdiff(argindex=1)

Returns the first derivative of this function.

5.7.9 C specific AST nodes (sympy.codegen.cnodes)

AST nodes specific to the C family of languages

class sympy.codegen.cnodes.**CommaOperator**

Represents the comma operator in C

class sympy.codegen.cnodes.**Label**

Label for use with e.g. goto statement.

Examples

```
>>> from sympy.codegen.cnodes import Label
>>> from sympy.printing.ccode import ccode
>>> print(ccode(Label('foo')))
foo:
```

class sympy.codegen.cnodes.**PostDecrement**

Represents the post-decrement operator

class sympy.codegen.cnodes.**PostIncrement**

Represents the post-increment operator

class sympy.codegen.cnodes.**PreDecrement**

Represents the pre-decrement operator

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cnodes import PreDecrement
>>> from sympy.printing.ccode import ccode
>>> ccode(PreDecrement(x))
'--(x)'
```

class sympy.codegen.cnodes.**PreIncrement**

Represents the pre-increment operator

`sympy.codegen.cnodes.alignof(arg)`
Generate of FunctionCall instance for calling 'alignof'

class `sympy.codegen.cnodes.goto`
Represents goto in C

`sympy.codegen.cnodes.sizeof(arg)`
Generate of FunctionCall instance for calling 'sizeof'

Examples

```
>>> from sympy.codegen.ast import real
>>> from sympy.codegen.cnodes import sizeof
>>> from sympy.printing.ccode import ccode
>>> ccode(sizeof(real))
'sizeof(double)'
```

class `sympy.codegen.cnodes.struct`
Represents a struct in C

class `sympy.codegen.cnodes.union`
Represents a union in C

5.7.10 C++ specific AST nodes (`sympy.codegen.cxxnodes`)

AST nodes specific to C++.

class `sympy.codegen.cxxnodes.using`
Represents a 'using' statement in C++

5.7.11 Fortran specific AST nodes (`sympy.codegen.fnodes`)

AST nodes specific to Fortran.

The functions defined in this module allows the user to express functions such as `dsign` as a SymPy function for symbolic manipulation.

class `sympy.codegen.fnodes.ArrayConstructor`
Represents an array constructor

Examples

```
>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import ArrayConstructor
>>> ac = ArrayConstructor([1, 2, 3])
>>> fcode(ac, standard=95, source_format='free')
'(/1, 2, 3/)'
>>> fcode(ac, standard=2003, source_format='free')
'[1, 2, 3]'
```

class `sympy.codegen.fnodes.Do`
Represents a Do loop in in Fortran

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import aug_assign, Print
>>> from sympy.codegen.fnodes import Do
>>> from sympy.printing import fcode
>>> i, n = symbols('i n', integer=True)
>>> r = symbols('r', real=True)
>>> body = [aug_assign(r, '+', 1/i), Print([i, r])]
>>> do1 = Do(body, i, 1, n)
>>> print(fcode(do1, source_format='free'))
do i = 1, n
    r = r + 1d0/i
    print *, i, r
end do
>>> do2 = Do(body, i, 1, n, 2)
>>> print(fcode(do2, source_format='free'))
do i = 1, n, 2
    r = r + 1d0/i
    print *, i, r
end do
```

class sympy.codegen.fnodes.Extent
Represents a dimension extent.

Examples

```
>>> from sympy.codegen.fnodes import Extent
>>> e = Extent(-3, 3) # -3, -2, -1, 0, 1, 2, 3
>>> from sympy.printing import fcode
>>> fcode(e, source_format='free')
'-3:3'
>>> from sympy.codegen.ast import Variable, real
>>> from sympy.codegen.fnodes import dimension, intent_out
>>> dim = dimension(e, e)
>>> arr = Variable('x', real, attrs=[dim, intent_out])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'real*8, dimension(-3:3, -3:3), intent(out) :: x'
```

class sympy.codegen.fnodes.FortranReturn
AST node explicitly mapped to a fortran “return”.

Because a return statement in fortran is different from C, and in order to aid reuse of our codegen ASTs the ordinary `.codegen.ast.Return` is interpreted as assignment to the result variable of the function. If one for some reason needs to generate a fortran RETURN statement, this node should be used.

Examples

```
>>> from sympy.codegen.fnodes import FortranReturn
>>> from sympy.printing import fcode
>>> fcode(FortranReturn('x'))
'    return x'
```

class sympy.codegen.fnodes.GoTo
Represents a goto statement in Fortran

Examples

```
>>> from sympy.codegen.fnodes import GoTo
>>> go = GoTo([10, 20, 30], 'i')
>>> from sympy.printing import fcode
>>> fcode(go, source_format='free')
'go to (10, 20, 30), i'
```

class sympy.codegen.fnodes.ImpliedDoLoop
Represents an implied do loop in Fortran

Examples

```
>>> from sympy import Symbol, fcode
>>> from sympy.codegen.fnodes import ImpliedDoLoop, ArrayConstructor
>>> i = Symbol('i', integer=True)
>>> idl = ImpliedDoLoop(i**3, i, -3, 3, 2) # -27, -1, 1, 27
>>> ac = ArrayConstructor([-28, idl, 28]) # -28, -27, -1, 1, 27, 28
>>> fcode(ac, standard=2003, source_format='free')
'[-28, (i**3, i = -3, 3, 2), 28]'
```

class sympy.codegen.fnodes.Module
Represents a module in Fortran

Examples

```
>>> from sympy.codegen.fnodes import Module
>>> from sympy.printing import fcode
>>> print(fcode(Module('signallib', ['implicit none'], []), source_format='free'))
module signallib
implicit none

contains

end module
```

class sympy.codegen.fnodes.Program
Represents a 'program' block in Fortran

Examples

```
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Program
>>> prog = Program('myprogram', [Print([42])])
>>> from sympy.printing import fcode
>>> print(fcode(prog, source_format='free'))
```

(continues on next page)

(continued from previous page)

```

program myprogram
  print *, 42
end program

```

class sympy.codegen.fnodes.Subroutine

Represents a subroutine in Fortran

Examples

```

>>> from sympy import symbols
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Subroutine
>>> from sympy.printing import fcode
>>> x, y = symbols('x y', real=True)
>>> sub = Subroutine('mysub', [x, y], [Print([x**2 + y**2, x*y])])
>>> print(fcode(sub, source_format='free', standard=2003))
subroutine mysub(x, y)
  real*8 :: x
  real*8 :: y
  print *, x**2 + y**2, x*y
end subroutine

```

class sympy.codegen.fnodes.SubroutineCall

Represents a call to a subroutine in Fortran

Examples

```

>>> from sympy.codegen.fnodes import SubroutineCall
>>> from sympy.printing import fcode
>>> fcode(SubroutineCall('mysub', 'x y'.split()))
'      call mysub(x, y)'

```

sympy.codegen.fnodes.allocated(array)

Creates an AST node for a function call to Fortran's "allocated(...)"

Examples

```

>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import allocated
>>> alloc = allocated('x')
>>> fcode(alloc, source_format='free')
'allocated(x)'

```

sympy.codegen.fnodes.array(symbol, dim, intent=None, **kwargs)

Convenience function for creating a Variable instance for a Fortran array

Parameters **symbol** : symbol**dim** : Attribute or iterable

If dim is an Attribute it need to have the name 'dimension'. If it is not an Attribute, then it is passed to `dimension()` (page 418) as *dim

intent : str

One of: 'in', 'out', 'inout' or None

****kwargs**:

Keyword arguments for Variable ('type' & 'value')

Examples

```
>>> from sympy.printing import fcode
>>> from sympy.codegen.ast import integer, real
>>> from sympy.codegen.fnodes import array
>>> arr = array('a', '*', 'in', type=integer)
>>> print(fcode(arr.as_Declaration(), source_format='free', standard=2003))
integer*4, dimension(*), intent(in) :: a
>>> x = array('x', [3, ':', ':'], intent='out', type=real)
>>> print(fcode(x.as_Declaration(value=1), source_format='free', standard=2003))
real*8, dimension(3, :, :), intent(out) :: x = 1
```

`sympy.codegen.fnodes.bind_C(name=None)`

Creates an Attribute `bind_C` with a name

Parameters `name` : str

Examples

```
>>> from sympy import Symbol
>>> from sympy.printing import fcode
>>> from sympy.codegen.ast import FunctionDefinition, real, Return, Variable
>>> from sympy.codegen.fnodes import array, sum_, size, bind_C
>>> a = Symbol('a', real=True)
>>> s = Symbol('s', integer=True)
>>> arr = array(a, dim=[s], intent='in')
>>> body = [Return((sum_(a**2)/s)**.5)]
>>> fd = FunctionDefinition(real, 'rms', [arr, s], body, attrs=[bind_C('rms')])
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a, s) bind(C, name="rms")
real*8, dimension(s), intent(in) :: a
integer*4 :: s
rms = sqrt(sum(a**2)/s)
end function
```

class `sympy.codegen.fnodes.cmplx`

Fortran complex conversion function.

`sympy.codegen.fnodes.dimension(*args)`

Creates a 'dimension' Attribute with (up to 7) extents.

Examples

```
>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import dimension, intent_in
>>> dim = dimension('2', ':') # 2 rows, runtime determined number of columns
```

(continues on next page)

(continued from previous page)

```
>>> from sympy.codegen.ast import Variable, integer
>>> arr = Variable('a', integer, attrs=[dim, intent_in])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'integer*4, dimension(2, :), intent(in) :: a'
```

class sympy.codegen.fnodes.dsign

Fortran sign intrinsic for double precision arguments.

class sympy.codegen.fnodes.isign

Fortran sign intrinsic for integer arguments.

class sympy.codegen.fnodes.kind

Fortran kind function.

sympy.codegen.fnodes.lbound(array, dim=None, kind=None)

Creates an AST node for a function call to Fortran's "lbound(...)"

Parameters array : Symbol or String

dim : expr

kind : expr

Examples

```
>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import lbound
>>> lb = lbound('arr', dim=2)
>>> fcode(lb, source_format='free')
'lbound(arr, 2)'
```

class sympy.codegen.fnodes.literal_dp

Fortran double precision real literal

Attributes

is_irrational	
is_rational	

class sympy.codegen.fnodes.literal_sp

Fortran single precision real literal

Attributes

is_irrational	
is_rational	

class sympy.codegen.fnodes.merge

Fortran merge function

sympy.codegen.fnodes.reshape(source, shape, pad=None, order=None)

Creates an AST node for a function call to Fortran's "reshape(...)"

Parameters **source** : Symbol or String

shape : ArrayExpr

`sympy.codegen.fnodes.shape`(source, kind=None)

Creates an AST node for a function call to Fortran's "shape(...)"

Parameters **source** : Symbol or String

kind : expr

Examples

```
>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import shape
>>> shp = shape('x')
>>> fcode(shp, source_format='free')
'shape(x)'
```

`sympy.codegen.fnodes.size`(array, dim=None, kind=None)

Creates an AST node for a function call to Fortran's "size(...)"

Examples

```
>>> from sympy import Symbol
>>> from sympy.printing import fcode
>>> from sympy.codegen.ast import FunctionDefinition, real, Return, Variable
>>> from sympy.codegen.fnodes import array, sum_, size
>>> a = Symbol('a', real=True)
>>> body = [Return((sum_(a**2)/size(a))**.5)]
>>> arr = array(a, dim=[':'], intent='in')
>>> fd = FunctionDefinition(real, 'rms', [arr], body)
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a)
real*8, dimension(:), intent(in) :: a
rms = sqrt(sum(a**2)*1d0/size(a))
end function
```

class `sympy.codegen.fnodes.use`

Represents a use statement in Fortran

Examples

```
>>> from sympy.codegen.fnodes import use
>>> from sympy.printing import fcode
>>> fcode(use('signallib'), source_format='free')
'use signallib'
>>> fcode(use('signallib', [('metric', 'snr')]), source_format='free')
'use signallib, metric => snr'
>>> fcode(use('signallib', only=['snr', 'convolution2d']), source_format='free')
'use signallib, only: snr, convolution2d'
```

class `sympy.codegen.fnodes.use_rename`

Represents a renaming in a use statement in Fortran

Examples

```
>>> from sympy.codegen.fnodes import use_rename, use
>>> from sympy.printing import fcode
>>> ren = use_rename("thingy", "convolution2d")
>>> print(fcode(ren, source_format='free'))
thingy => convolution2d
>>> full = use('signallib', only=['snr', ren])
>>> print(fcode(full, source_format='free'))
use signallib, only: snr, thingy => convolution2d
```

5.7.12 Algorithms (sympy.codegen.algorithms)

`sympy.codegen.algorithms.newtons_method(expr, wrt, atol=1e-12, delta=None, debug=False, itermax=None, counter=None)`

Generates an AST for Newton-Raphson method (a root-finding algorithm).

Returns an abstract syntax tree (AST) based on `sympy.codegen.ast` for Newton's method of root-finding.

Parameters **expr** : expression

wrt : Symbol

With respect to, i.e. what is the variable.

atol : number or expr

Absolute tolerance (stopping criterion)

delta : Symbol

Will be a Dummy if None.

debug : bool

Whether to print convergence information during iterations

itermax : number or expr

Maximum number of iterations.

counter : Symbol

Will be a Dummy if None.

References

[R26] (page 1857)

Examples

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import Assignment
>>> from sympy.codegen.algorithms import newtons_method
>>> x, dx, atol = symbols('x dx atol')
>>> expr = cos(x) - x**3
```

(continues on next page)

(continued from previous page)

```
>>> algo = newtons_method(expr, x, atol, dx)
>>> algo.has(Assignment(dx, -expr/expr.diff(x)))
True
```

`sympy.codegen.algorithms.newtons_method_function(expr, wrt, params=None, func_name='newton', attrs=(), **kwargs)`

Generates an AST for a function implementing the Newton-Raphson method.

Parameters **expr** : expression

wrt : Symbol

With respect to, i.e. what is the variable

params : iterable of symbols

Symbols appearing in `expr` that are taken as constants during the iterations (these will be accepted as parameters to the generated function).

func_name : str

Name of the generated function.

attrs : Tuple

Attribute instances passed as `attrs` to `FunctionDefinition`.

****kwargs** :

Keyword arguments passed to `sympy.codegen.algorithms.newtons_method()` (page 421).

See also:

-

Examples

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.algorithms import newtons_method_function
>>> from sympy.codegen.pyutils import render_as_module
>>> from sympy.core.compatibility import exec_
>>> x = symbols('x')
>>> expr = cos(x) - x**3
>>> func = newtons_method_function(expr, x)
>>> py_mod = render_as_module(func) # source code as string
>>> namespace = {}
>>> exec_(py_mod, namespace, namespace)
>>> res = eval('newton(0.5)', namespace)
>>> abs(res - 0.865474033102) < 1e-12
True
```

5.7.13 Python utilities (sympy.codegen.pyutils)

`sympy.codegen.pyutils.render_as_module(content)`

Renders python code as a module (with the required imports)

5.7.14 C utilities (sympy.codegen.cutils)

```
sympy.codegen.cutils.render_as_source_file(content, Printer=<class
                                         'sympy.printing.ccode.C99CodePrinter'>,
                                         settings=None)
    Renders a C source file (with required #include statements)
```

5.7.15 Fortran utilities (sympy.codegen.futils)

```
sympy.codegen.futils.render_as_module(definitions, name, declarations=(),
                                       printer_settings=None)
    Creates a Module instance and renders it as a string.
```

This generates Fortran source code for a module with the correct use statements.

Parameters **definitions** : iterable

Passed to `sympy.codegen.fnodes.Module` (page 416).

name : str

Passed to `sympy.codegen.fnodes.Module` (page 416).

declarations : iterable

Passed to `sympy.codegen.fnodes.Module` (page 416). It will be extended with use statements, 'implicit none' and public list generated from definitions.

printer_settings : dict

Passed to `FCodePrinter` (default: `{'standard': 2003, 'source_format': 'free'}`).

5.8 Numeric Computation

Symbolic computer algebra systems like SymPy facilitate the construction and manipulation of mathematical expressions. Unfortunately when it comes time to evaluate these expressions on numerical data, symbolic systems often have poor performance.

Fortunately SymPy offers a number of easy-to-use hooks into other numeric systems, allowing you to create mathematical expressions in SymPy and then ship them off to the numeric system of your choice. This page documents many of the options available including the math library, the popular array computing package numpy, code generation in Fortran or C, and the use of the array compiler Theano.

5.8.1 Subs/evalf

Subs is the slowest but simplest option. It runs at SymPy speeds. The `.subs(...).evalf()` method can substitute a numeric value for a symbolic one and then evaluate the result within SymPy.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

(continues on next page)

(continued from previous page)

```
>>> expr.evalf(subs={x: 3.14})
0.000507214304613640
```

This method is slow. You should use this method production only if performance is not an issue. You can expect `.subs` to take tens of microseconds. It can be useful while prototyping or if you just want to see a value once.

5.8.2 Lambdify

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. It is used as follows:

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr)
>>> f(3.14)
0.000507214304614
```

Here `lambdify` makes a function that computes $f(x) = \sin(x)/x$. By default `lambdify` relies on implementations in the `math` standard library. This numerical evaluation takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs` method. This is the speed difference between SymPy and raw Python.

`Lambdify` can leverage a variety of numerical backends. By default it uses the `math` library. However it also supports `mpmath` and most notably, `numpy`. Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr, "numpy")
```

```
>>> import numpy
>>> data = numpy.linspace(1, 10, 10000)
>>> f(data)
[ 0.84147098  0.84119981  0.84092844 ... -0.05426074 -0.05433146
 -0.05440211]
```

If you have array-based data this can confer a considerable speedup, on the order of 10 nanoseconds per element. Unfortunately `numpy` incurs some start-up time and introduces an overhead of a few microseconds.

5.8.3 uFuncify

The `autowrap` module contains methods that help in efficient computation.

- [autowrap](#) (page 386) method for compiling code generated by the [codegen](#) (page 380) module, and wrap the binary for use in python.
- [binary_function](#) (page 388) method automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with `implemented_function()`.

- [ufuncify](#) (page 388) generates a binary function that supports broadcasting on numpy arrays using different backends that are faster as compared to `subs/evalf` and `lambdify`.

The API reference of all the above is listed here: [sympy.utilities.autowrap\(\)](#) (page 1391).

5.8.4 Theano

SymPy has a strong connection with [Theano](#), a mathematical array compiler. SymPy expressions can be easily translated to Theano graphs and then compiled using the Theano compiler chain.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```
>>> from sympy.printing.theanocode import theano_function
>>> f = theano_function([x], [expr])
```

If array broadcasting or types are desired then Theano requires this extra information

```
>>> f = theano_function([x], [expr], dims={x: 1}, dtypes={x: 'float64'})
```

Theano has a more sophisticated code generation system than SymPy's C/Fortran code printers. Among other things it handles common sub-expressions and compilation onto the GPU. Theano also supports SymPy Matrix and Matrix Expression objects.

5.8.5 So Which Should I Use?

The options here were listed in order from slowest and least dependencies to fastest and most dependencies. For example, if you have Theano installed then that will often be the best choice. If you don't have Theano but do have f2py then you should use `ufuncify`.

Tool	Speed	Qualities	Dependencies
<code>subs/evalf</code>	50us	Simple	None
<code>lambdify</code>	1us	Scalar functions	math
<code>lambdify-numpy</code>	10ns	Vector functions	numpy
<code>ufuncify</code>	10ns	Complex vector expressions	f2py, Cython
Theano	10ns	Many outputs, CSE, GPUs	Theano

5.9 Functions Module

All functions support the methods documented below, inherited from [sympy.core.function.Function](#) (page 183).

class `sympy.core.function.Function`

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

Examples

First example shows how to use Function as a constructor for undefined function classes:

```
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

Assumptions can be passed to Function.

```
>>> f_real = Function('f', real=True)
>>> f_real(x).is_real
True
```

Note that assumptions on a function are unrelated to the assumptions on the variable it is called on. If you want to add a relationship, subclass Function and define the appropriate `_eval_is_assumption` methods.

In the following example Function is used as a base class for `my_func` that represents a mathematical function `my_func`. Suppose that it is well known, that `my_func(0)` is 1 and `my_func` at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that `my_func(x)` is real exactly when `x` is real. Here is an implementation that honours those requirements:

```
>>> from sympy import Function, S, oo, I, sin
>>> class my_func(Function):
...
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x is S.Zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...         def _eval_is_real(self):
...             return self.args[0].is_real
...
>>> x = S('x')
>>> my_func(0) + sin(0)
1
>>> my_func(oo)
0
>>> my_func(3.54).n() # Not yet implemented for my_func.
my_func(3.54)
>>> my_func(I).is_real
False
```


In order for `my_func` to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then `nargs` must be defined, e.g. if `my_func` can take one or two arguments then,

```
>>> class my_func(Function):
...     nargs = (1, 2)
...
>>>
```

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

fdiff(argindex=1)

Returns the first derivative of the function.

is_commutative

Returns whether the function is commutative.

5.9.1 Contents

Elementary

This module implements elementary functions such as trigonometric, hyperbolic, and `sqrt`, as well as functions like `Abs`, `Max`, `Min` etc.

`sympy.functions.elementary.complexes`

`re`

class `sympy.functions.elementary.complexes.re`

Returns real part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

See also:

[im](#) (page 428)

Examples

```
>>> from sympy import re, im, I, E
>>> from sympy.abc import x, y
>>> re(2*E)
2*E
>>> re(2*I + 17)
17
>>> re(2*I)
0
>>> re(im(x) + x*I + 2)
2
```

as_real_imag(deep=True, **hints)
Returns the real number with a zero imaginary part.

im

class sympy.functions.elementary.complexes.**im**

Returns imaginary part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

See also:

[re](#) (page 427)

Examples

```
>>> from sympy import re, im, E, I
>>> from sympy.abc import x, y
>>> im(2*E)
0
>>> re(2*I + 17)
17
>>> im(x*I)
re(x)
>>> im(re(x) + y)
im(y)
```

as_real_imag(deep=True, **hints)
Return the imaginary part with a zero real part.

Examples

```
>>> from sympy.functions import im
>>> from sympy import I
>>> im(2 + 3*I).as_real_imag()
(3, 0)
```

sign

class sympy.functions.elementary.complexes.**sign**

Returns the complex sign of an expression:

If the expression is real the sign will be:

- 1 if expression is positive
- 0 if expression is equal to zero
- -1 if expression is negative

If the expression is imaginary the sign will be:

- I if `im(expression)` is positive

- $-I$ if $\text{im}(\text{expression})$ is negative

Otherwise an unevaluated expression will be returned. When evaluated, the result (in general) will be $\cos(\arg(\text{expr})) + I\sin(\arg(\text{expr}))$.

See also:

[Abs](#) (page 429), [conjugate](#) (page 430)

Examples

```
>>> from sympy.functions import sign
>>> from sympy.core.numbers import I
```

```
>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> _evalf()
0.707106781186548 + 0.707106781186548*I
```

Abs

class sympy.functions.elementary.complexes.**Abs**

Return the absolute value of the argument.

This is an extension of the built-in function `abs()` to accept symbolic values. If you pass a SymPy expression to the built-in `abs()`, it will pass it automatically to `Abs()`.

See also:

[sign](#) (page 428), [conjugate](#) (page 430)

Examples

```
>>> from sympy import Abs, Symbol, S
>>> Abs(-1)
1
>>> x = Symbol('x', real=True)
>>> Abs(-x)
Abs(x)
>>> Abs(x**2)
x**2
>>> abs(-x) # The Python built-in
Abs(x)
```

Note that the Python built-in will return either an `Expr` or `int` depending on the argument:

```
>>> type(abs(-1))
<... 'int'>
>>> type(abs(S.NegativeOne))
<class 'sympy.core.numbers.One'>
```

Abs will always return a sympy object.

fdiff(argindex=1)

Get the first derivative of the argument to Abs().

Examples

```
>>> from sympy.abc import x
>>> from sympy.functions import Abs
>>> Abs(-x).fdiff()
sign(x)
```

arg

class sympy.functions.elementary.complexes.**arg**

Returns the argument (in radians) of a complex number. For a positive number, the argument is always 0.

Examples

```
>>> from sympy.functions import arg
>>> from sympy import I, sqrt
>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
```

conjugate

class sympy.functions.elementary.complexes.**conjugate**

Returns the *complexconjugate* Ref[1] of an argument. In mathematics, the complex conjugate of a complex number is given by changing the sign of the imaginary part.

Thus, the conjugate of the complex number $a + ib$ (where a and b are real numbers) is $a - ib$

See also:

[sign](#) (page 428), [Abs](#) (page 429)

References

[R116] (page 1857)

Examples

```
>>> from sympy import conjugate, I
>>> conjugate(2)
2
>>> conjugate(I)
-I
```

polar_lift

class sympy.functions.elementary.complexes.polar_lift

Lift argument to the Riemann surface of the logarithm, using the standard branch.

```
>>> from sympy import Symbol, polar_lift, I
>>> p = Symbol('p', polar=True)
>>> x = Symbol('x')
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)
```

```
>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

sympy.functions.elementary.exponential.exp_polar, [periodic_argument](#)
(page 431)

periodic_argument

class sympy.functions.elementary.complexes.periodic_argument

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period P , always return a value in $(-P/2, P/2]$, by using $\exp(P*I) == 1$.

```
>>> from sympy import exp, exp_polar, periodic_argument, unbranched_argument
>>> from sympy import I, pi
>>> unbranched_argument(exp(5*I*pi))
pi
>>> unbranched_argument(exp_polar(5*I*pi))
5*pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0
```

See also:

`sympy.functions.elementary.exponential.exp_polar`

polar_lift (page 431) Lift argument to the Riemann surface of the logarithm

`principal_branch` (page 432)

principal_branch

class `sympy.functions.elementary.complexes.principal_branch`

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

This is a function of two arguments. The first argument is a polar number z , and the second one a positive real number of infinity, p . The result is “ $z \bmod \exp\_polar(I*p)$ ”.

```
>>> from sympy import exp_polar, principal_branch, oo, I, pi
>>> from sympy.abc import z
>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)
```

See also:

`sympy.functions.elementary.exponential.exp_polar`

polar_lift (page 431) Lift argument to the Riemann surface of the logarithm

`periodic_argument` (page 431)

sympy.functions.elementary.trigonometric**Trigonometric Functions****sin**

class `sympy.functions.elementary.trigonometric.sin`

The sine function.

Returns the sine of x (measured in radians).

See also:

`csc` (page 436), `cos` (page 433), `sec` (page 435), `tan` (page 434), `cot` (page 435), `asin` (page 437), `acsc` (page 441), `acos` (page 438), `asec` (page 440), `atan` (page 439), `acot` (page 439), `atan2` (page 441)

Notes

This function will evaluate automatically in the case x/π is some rational number [R120] (page 1857). For example, if x is a multiple of π , $\pi/2$, $\pi/3$, $\pi/4$ and $\pi/6$.

References

[R117] (page 1857), [R118] (page 1857), [R119] (page 1857), [R120] (page 1857)

Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x
>>> sin(x**2).diff(x)
2*x*cos(x**2)
>>> sin(1).diff(x)
0
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
1/2
>>> sin(pi/12)
-sqrt(2)/4 + sqrt(6)/4
```

Members

cos

class sympy.functions.elementary.trigonometric.cos

The cosine function.

Returns the cosine of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

See [sin\(\)](#) (page 432) for notes about automatic evaluation.

References

[R121] (page 1857), [R122] (page 1857), [R123] (page 1857)

Examples

```
>>> from sympy import cos, pi
>>> from sympy.abc import x
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
```

(continues on next page)

(continued from previous page)

```
>>> cos(1).diff(x)
0
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
>>> cos(pi/12)
sqrt(2)/4 + sqrt(6)/4
```

Members

tan

class `sympy.functions.elementary.trigonometric.tan`

The tangent function.

Returns the tangent of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

See [sin\(\)](#) (page 432) for notes about automatic evaluation.

References

[R124] (page 1857), [R125] (page 1857), [R126] (page 1857)

Examples

```
>>> from sympy import tan, pi
>>> from sympy.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(1).diff(x)
0
>>> tan(pi/8).expand()
-1 + sqrt(2)
```

Members

cot

class `sympy.functions.elementary.trigonometric.cot`

The cotangent function.

Returns the cotangent of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

See [sin\(\)](#) (page 432) for notes about automatic evaluation.

References

[R127] (page 1857), [R128] (page 1857), [R129] (page 1857)

Examples

```

>>> from sympy import cot, pi
>>> from sympy.abc import x
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
>>> cot(1).diff(x)
0
>>> cot(pi/12)
sqrt(3) + 2

```

Members

sec

class `sympy.functions.elementary.trigonometric.sec`

The secant function.

Returns the secant of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

See [sin\(\)](#) (page 432) for notes about automatic evaluation.

References

[R130] (page 1857), [R131] (page 1857), [R132] (page 1858)

Examples

```
>>> from sympy import sec
>>> from sympy.abc import x
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
>>> sec(1).diff(x)
0
```

Members

csc

class sympy.functions.elementary.trigonometric.csc

The cosecant function.

Returns the cosecant of x (measured in radians).

See also:

[sin](#) (page 432), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

See [sin\(\)](#) (page 432) for notes about automatic evaluation.

References

[R133] (page 1858), [R134] (page 1858), [R135] (page 1858)

Examples

```
>>> from sympy import csc
>>> from sympy.abc import x
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
>>> csc(1).diff(x)
0
```

Members

sinc

class `sympy.functions.elementary.trigonometric.sinc`
 Represents unnormalized sinc function

References

[R136] (page 1858)

Examples

```
>>> from sympy import sinc, oo, jn, Product, Symbol
>>> from sympy.abc import x
>>> sinc(x)
sinc(x)
```

- Automated Evaluation

```
>>> sinc(0)
1
>>> sinc(oo)
0
```

- Differentiation

```
>>> sinc(x).diff()
(x*cos(x) - sin(x))/x**2
```

- Series Expansion

```
>>> sinc(x).series()
1 - x**2/6 + x**4/120 + O(x**6)
```

- As zero'th order spherical Bessel Function

```
>>> sinc(x).rewrite(jn)
jn(0, x)
```

Members

Trigonometric Inverses

asin

class `sympy.functions.elementary.trigonometric.asin`
 The inverse sine function.
 Returns the arcsine of x in radians.

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

`asin(x)` will evaluate automatically in the cases `oo`, `-oo`, `0`, `1`, `-1` and for some instances when the result is a rational multiple of `pi` (see the `eval` class method).

References

[R137] (page 1858), [R138] (page 1858), [R139] (page 1858)

Examples

```
>>> from sympy import asin, oo, pi
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2
```

Members

acos

class `sympy.functions.elementary.trigonometric.acos`

The inverse cosine function.

Returns the arc cosine of `x` (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

`acos(x)` will evaluate automatically in the cases `oo`, `-oo`, `0`, `1`, `-1`.

`acos(zoo)` evaluates to `zoo` (see note in `:py:class'sympy.functions.elementary.trigonometric.asec'`)

References

[R140] (page 1858), [R141] (page 1858), [R142] (page 1858)

Examples

```
>>> from sympy import acos, oo, pi
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I
```

Members

atan

class sympy.functions.elementary.trigonometric.**atan**

The inverse tangent function.

Returns the arc tangent of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [acot](#) (page 439), [atan2](#) (page 441)

Notes

atan(x) will evaluate automatically in the cases oo, -oo, 0, 1, -1.

References

[R143] (page 1858), [R144] (page 1858), [R145] (page 1858)

Examples

```
>>> from sympy import atan, oo, pi
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2
```

Members

acot

class sympy.functions.elementary.trigonometric.**acot**

The inverse cotangent function.

Returns the arc cotangent of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [atan2](#) (page 441)

References

[\[R146\]](#) (page 1858), [\[R147\]](#) (page 1858), [\[R148\]](#) (page 1858)

Members

asec

class `sympy.functions.elementary.trigonometric.asec`

The inverse secant function.

Returns the arc secant of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

`asec(x)` will evaluate automatically in the cases ∞ , $-\infty$, 0 , 1 , -1 .

`asec(x)` has branch cut in the interval $[-1, 1]$. For complex arguments, it can be defined [\[R152\]](#) (page 1858) as

$$\sec^{-1}(z) = -i * (\log(\sqrt{1 - z^2} + 1)/z)$$

At $x = 0$, for positive branch cut, the limit evaluates to ∞ . For negative branch cut, the limit

$$\lim_{z \rightarrow 0} -i * (\log(-\sqrt{1 - z^2} + 1)/z)$$

simplifies to $-i * \log(z/2 + O(z^3))$ which ultimately evaluates to ∞ .

As `asec(x) = asec(1/x)`, a similar argument can be given for `acos(x)`.

References

[\[R149\]](#) (page 1858), [\[R150\]](#) (page 1858), [\[R151\]](#) (page 1858), [\[R152\]](#) (page 1858)

Examples

```
>>> from sympy import asec, oo, pi
>>> asec(1)
0
>>> asec(-1)
pi
```

Members

acsc

class sympy.functions.elementary.trigonometric.**acsc**

The inverse cosecant function.

Returns the arc cosecant of x (measured in radians).

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439), [atan2](#) (page 441)

Notes

$\text{acsc}(x)$ will evaluate automatically in the cases ∞ , $-\infty$, 0, 1, -1.

References

[R153] (page 1858), [R154] (page 1858), [R155] (page 1858)

Examples

```
>>> from sympy import acsc, oo, pi
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
```

Members

atan2

class sympy.functions.elementary.trigonometric.**atan2**

The function $\text{atan2}(y, x)$ computes $\text{atan}(y/x)$ taking two arguments y and x . Signs of both y and x are considered to determine the appropriate quadrant of $\text{atan}(y/x)$. The

range is $(-\pi, \pi]$. The complete definition reads as follows:

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Attention: Note the role reversal of both arguments. The y -coordinate is the first argument and the x -coordinate the second.

See also:

[sin](#) (page 432), [csc](#) (page 436), [cos](#) (page 433), [sec](#) (page 435), [tan](#) (page 434), [cot](#) (page 435), [asin](#) (page 437), [acsc](#) (page 441), [acos](#) (page 438), [asec](#) (page 440), [atan](#) (page 439), [acot](#) (page 439)

References

[\[R156\]](#) (page 1858), [\[R157\]](#) (page 1858), [\[R158\]](#) (page 1858)

Examples

Going counter-clock wise around the origin we find the following angles:

```
>>> from sympy import atan2
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary `atan` function for the point $(x, y) = (-1, 1)$

```
>>> from sympy import atan, S
>>> atan(S(1) / -1)
-pi/4
>>> atan2(1, -1)
3*pi/4
```

where only the `atan2` function returns what we expect. We can differentiate the function with respect to both arguments:


```
>>> from sympy import diff
>>> from sympy.abc import x, y
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)
```

```
>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the `atan2` function in terms of complex logarithms:

```
>>> from sympy import log
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of (*atan*):

```
>>> from sympy import atan
>>> atan2(y, x).rewrite(atan)
2*atan(y/(x + sqrt(x**2 + y**2)))
```

but note that this form is undefined on the negative real axis.

Members

`sympy.functions.elementary.hyperbolic`

Hyperbolic Functions

HyperbolicFunction

class `sympy.functions.elementary.hyperbolic.HyperbolicFunction`
Base class for hyperbolic functions.

See also:

[sinh](#) (page 443), [cosh](#) (page 444), [tanh](#) (page 444), [coth](#) (page 444)

Members

`sinh`

class `sympy.functions.elementary.hyperbolic.sinh`
The hyperbolic sine function, $\frac{e^x - e^{-x}}{2}$.

- `sinh(x)` -> Returns the hyperbolic sine of `x`

See also:

[cosh](#) (page 444), [tanh](#) (page 444), [asinh](#) (page 445)

Members

cosh

class sympy.functions.elementary.hyperbolic.cosh

The hyperbolic cosine function, $\frac{e^x + e^{-x}}{2}$.

- cosh(x) -> Returns the hyperbolic cosine of x

See also:

[sinh](#) (page 443), [tanh](#) (page 444), [acosh](#) (page 445)

Members

tanh

class sympy.functions.elementary.hyperbolic.tanh

The hyperbolic tangent function, $\frac{\sinh(x)}{\cosh(x)}$.

- tanh(x) -> Returns the hyperbolic tangent of x

See also:

[sinh](#) (page 443), [cosh](#) (page 444), [atanh](#) (page 445)

Members

coth

class sympy.functions.elementary.hyperbolic.coth

The hyperbolic cotangent function, $\frac{\cosh(x)}{\sinh(x)}$.

- coth(x) -> Returns the hyperbolic cotangent of x

Members

sech

class sympy.functions.elementary.hyperbolic.sech

The hyperbolic secant function, $\frac{2}{e^x + e^{-x}}$

- sech(x) -> Returns the hyperbolic secant of x

See also:

[sinh](#) (page 443), [cosh](#) (page 444), [tanh](#) (page 444), [coth](#) (page 444), [csch](#) (page 444), [asinh](#) (page 445), [acosh](#) (page 445)

Members

csch

class sympy.functions.elementary.hyperbolic.csch

The hyperbolic cosecant function, $\frac{2}{e^x - e^{-x}}$

- csch(x) -> Returns the hyperbolic cosecant of x

See also:

[sinh](#) (page 443), [cosh](#) (page 444), [tanh](#) (page 444), [sech](#) (page 444), [asinh](#) (page 445), [acosh](#) (page 445)

Members**Hyperbolic Inverses****asinh**

class `sympy.functions.elementary.hyperbolic.asinh`

The inverse hyperbolic sine function.

- `asinh(x)` -> Returns the inverse hyperbolic sine of x

See also:

[acosh](#) (page 445), [atanh](#) (page 445), [sinh](#) (page 443)

Members**acosh**

class `sympy.functions.elementary.hyperbolic.acosh`

The inverse hyperbolic cosine function.

- `acosh(x)` -> Returns the inverse hyperbolic cosine of x

See also:

[asinh](#) (page 445), [atanh](#) (page 445), [cosh](#) (page 444)

Members**atanh**

class `sympy.functions.elementary.hyperbolic.atanh`

The inverse hyperbolic tangent function.

- `atanh(x)` -> Returns the inverse hyperbolic tangent of x

See also:

[asinh](#) (page 445), [acosh](#) (page 445), [tanh](#) (page 444)

Members**acoth**

class `sympy.functions.elementary.hyperbolic.acoth`

The inverse hyperbolic cotangent function.

- `acoth(x)` -> Returns the inverse hyperbolic cotangent of x

Members

asech

class sympy.functions.elementary.hyperbolic.asech

The inverse hyperbolic secant function.

- asech(x) -> Returns the inverse hyperbolic secant of x

See also:

[asinh](#) (page 445), [atanh](#) (page 445), [cosh](#) (page 444), [acoth](#) (page 445)

References

[R159] (page 1858), [R160] (page 1858), [R161] (page 1858)

Examples

```
>>> from sympy import asech, sqrt, S
>>> from sympy.abc import x
>>> asech(x).diff(x)
-1/(x*sqrt(-x**2 + 1))
>>> asech(1).diff(x)
0
>>> asech(1)
0
>>> asech(S(2))
I*pi/3
>>> asech(-sqrt(2))
3*I*pi/4
>>> asech((sqrt(6) - sqrt(2)))
I*pi/12
```

Members

acsch

class sympy.functions.elementary.hyperbolic.acsch

The inverse hyperbolic cosecant function.

- acsch(x) -> Returns the inverse hyperbolic cosecant of x

References

[R162] (page 1858), [R163] (page 1858), [R164] (page 1858)

Examples

```

>>> from sympy import acsch, sqrt, S
>>> from sympy.abc import x
>>> acsch(x).diff(x)
-1/(x**2*sqrt(1 + x**(-2)))
>>> acsch(1).diff(x)
0
>>> acsch(1)
log(1 + sqrt(2))
>>> acsch(S.ImaginaryUnit)
-I*pi/2
>>> acsch(-2*S.ImaginaryUnit)
I*pi/6
>>> acsch(S.ImaginaryUnit*(sqrt(6) - sqrt(2)))
-5*I*pi/12

```

Members

sympy.functions.elementary.integers

ceiling

class sympy.functions.elementary.integers.ceiling

Ceiling is a univariate function which returns the smallest integer value not less than its argument. This implementation generalizes ceiling to complex numbers by taking the ceiling of the real and imaginary parts separately.

See also:

[sympy.functions.elementary.integers.floor](#) (page 448)

References

[R165] (page 1858), [R166] (page 1858)

Examples

```

>>> from sympy import ceiling, E, I, S, Float, Rational
>>> ceiling(17)
17
>>> ceiling(Rational(23, 10))
3
>>> ceiling(2*E)
6
>>> ceiling(-Float(0.567))
0
>>> ceiling(I/2)
I
>>> ceiling(S(5)/2 + 5*I/2)
3 + 3*I

```

Members

floor

class `sympy.functions.elementary.integers.floor`

Floor is a univariate function which returns the largest integer value not greater than its argument. This implementation generalizes floor to complex numbers by taking the floor of the real and imaginary parts separately.

See also:

`sympy.functions.elementary.integers.ceiling` (page 447)

References

[R167] (page 1858), [R168] (page 1858)

Examples

```
>>> from sympy import floor, E, I, S, Float, Rational
>>> floor(17)
17
>>> floor(Rational(23, 10))
2
>>> floor(2*E)
5
>>> floor(-Float(0.567))
-1
>>> floor(-I/2)
-I
>>> floor(S(5)/2 + 5*I/2)
2 + 2*I
```

Members

RoundFunction

class `sympy.functions.elementary.integers.RoundFunction`

The base class for rounding functions.

frac

class `sympy.functions.elementary.integers.frac`

Represents the fractional part of x

For real numbers it is defined [R169] (page 1858) as

$$x - \lfloor x \rfloor$$

See also:

`sympy.functions.elementary.integers.floor` (page 448), `sympy.functions.elementary.integers.ceiling` (page 447)

References

[R169] (page 1858), [R170] (page 1859)

Examples

```
>>> from sympy import Symbol, frac, Rational, floor, ceiling, I
>>> frac(Rational(4, 3))
1/3
>>> frac(-Rational(4, 3))
2/3
```

returns zero for integer arguments

```
>>> n = Symbol('n', integer=True)
>>> frac(n)
0
```

rewrite as floor

```
>>> x = Symbol('x')
>>> frac(x).rewrite(floor)
x - floor(x)
```

for complex arguments

```
>>> r = Symbol('r', real=True)
>>> t = Symbol('t', real=True)
>>> frac(t + I*r)
I*frac(r) + frac(t)
```

sympy.functions.elementary.exponential

exp

class sympy.functions.elementary.exponential.exp

The exponential function, e^x .

See also:

[log](#) (page 450)

Members

LambertW

class sympy.functions.elementary.exponential.LambertW

The Lambert W function $W(z)$ is defined as the inverse function of $w \exp(w)$ [R171] (page 1859).

In other words, the value of $W(z)$ is such that $z = W(z) \exp(W(z))$ for any complex number z . The Lambert W function is a multivalued function with infinitely many branches $W_k(z)$, indexed by $k \in \mathbb{Z}$. Each branch gives a different solution w of the equation $z = w \exp(w)$.

The Lambert W function has two partially real branches: the principal branch ($k = 0$) is real for real $z > -1/e$, and the $k = -1$ branch is real for $-1/e < z < 0$. All branches except $k = 0$ have a logarithmic singularity at $z = 0$.

References

[R171] (page 1859)

Examples

```
>>> from sympy import LambertW
>>> LambertW(1.2)
0.635564016364870
>>> LambertW(1.2, -1).n()
-1.34747534407696 - 4.41624341514535*I
>>> LambertW(-1).is_real
False
```

Members

log

class sympy.functions.elementary.exponential.log

The natural logarithm function $\ln(x)$ or $\log(x)$. Logarithms are taken with the natural base, e . To get a logarithm of a different base b , use $\log(x, b)$, which is essentially short-hand for $\log(x)/\log(b)$.

See also:

[exp](#) (page 449)

Members

sympy.functions.elementary.piecewise

ExprCondPair

class sympy.functions.elementary.piecewise.ExprCondPair

Represents an expression, condition pair.

Members

Piecewise

class sympy.functions.elementary.piecewise.Piecewise

Represents a piecewise function.

Usage:

Piecewise((expr,cond), (expr,cond), ...)

- Each argument is a 2-tuple defining an expression and condition

- The conds are evaluated in turn returning the first that is True. If any of the evaluated conds are not determined explicitly False, e.g. $x < 1$, the function is returned in symbolic form.
- If the function is evaluated at a place where all conditions are False, a `ValueError` exception will be raised.
- Pairs where the cond is explicitly False, will be removed.

See also:

[piecewise_fold](#) (page 451), [ITE](#)

Examples

```
>>> from sympy import Piecewise, log, ITE, piecewise_fold
>>> from sympy.abc import x, y
>>> f = x**2
>>> g = log(x)
>>> p = Piecewise((0, x < -1), (f, x <= 1), (g, True))
>>> p.subs(x,1)
1
>>> p.subs(x,5)
log(5)
```

Booleans can contain Piecewise elements:

```
>>> cond = (x < y).subs(x, Piecewise((2, x < 0), (3, True))); cond
Piecewise((2, x < 0), (3, True)) < y
```

The folded version of this results in a Piecewise whose expressions are Booleans:

```
>>> folded_cond = piecewise_fold(cond); folded_cond
Piecewise((2 < y, x < 0), (3 < y, True))
```

When a Boolean containing Piecewise (like `cond`) or a Piecewise with Boolean expressions (like `folded_cond`) is used as a condition, it is converted to an equivalent ITE object:

```
>>> Piecewise((1, folded_cond))
Piecewise((1, ITE(x < 0, y > 2, y > 3)))
```

When a condition is an ITE, it will be converted to a simplified Boolean expression:

```
>>> piecewise_fold(_)
Piecewise((1, ((x >= 0) | (y > 2)) & ((y > 3) | (x < 0))))
```

Members

`sympy.functions.elementary.piecewise.piecewise_fold(expr)`

Takes an expression containing a piecewise function and returns the expression in piecewise form. In addition, any ITE conditions are rewritten in negation normal form and simplified.

See also:

[Piecewise](#) (page 450)

Examples

```
>>> from sympy import Piecewise, piecewise_fold, sympify as S
>>> from sympy.abc import x
>>> p = Piecewise((x, x < 1), (1, S(1) <= x))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, True))
```

sympy.functions.elementary.miscellaneous

IdentityFunction

class sympy.functions.elementary.miscellaneous.IdentityFunction

The identity function

Examples

```
>>> from sympy import Id, Symbol
>>> x = Symbol('x')
>>> Id(x)
x
```

Members

Min

class sympy.functions.elementary.miscellaneous.Min

Return, if possible, the minimum value of the list. It is named Min and not min to avoid conflicts with the built-in function min.

See also:

Max (page 453) find maximum values

Examples

```
>>> from sympy import Min, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Min(x, -2)
Min(x, -2)
>>> Min(x, -2).subs(x, 3)
-2
>>> Min(p, -3)
-3
>>> Min(x, y)
Min(x, y)
```

(continues on next page)

(continued from previous page)

```
>>> Min(n, 8, p, -7, p, oo)
Min(n, -7)
```

Members

Max

class sympy.functions.elementary.miscellaneous.Max

Return, if possible, the maximum value of the list.

When number of arguments is equal one, then return this argument.

When number of arguments is equal two, then return, if possible, the value from (a, b) that is $\geq$ the other.

In common case, when the length of list greater than 2, the task is more complicated. Return only the arguments, which are greater than others, if it is possible to determine directional relation.

If is not possible to determine such a relation, return a partially evaluated result.

Assumptions are used to make the decision too.

Also, only comparable arguments are permitted.

It is named Max and not max to avoid conflicts with the built-in function max.

See also:

Min (page 452) find minimum values

References

[R172] (page 1859), [R173] (page 1859)

Examples

```
>>> from sympy import Max, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Max(x, -2)
Max(x, -2)
>>> Max(x, -2).subs(x, 3)
3
>>> Max(p, -2)
p
>>> Max(x, y)
Max(x, y)
>>> Max(x, y) == Max(y, x)
True
>>> Max(x, Max(y, z))
Max(x, y, z)
```

(continues on next page)

(continued from previous page)

```
>>> Max(n, 8, p, 7, -oo)
Max(8, p)
>>> Max(1, x, oo)
oo
```

- Algorithm

The task can be considered as searching of supremums in the directed complete partial orders [R172] (page 1859).

The source values are sequentially allocated by the isolated subsets in which supremums are searched and result as Max arguments.

If the resulted supremum is single, then it is returned.

The isolated subsets are the sets of values which are only the comparable with each other in the current set. E.g. natural numbers are comparable with each other, but not comparable with the x symbol. Another example: the symbol x with negative assumption is comparable with a natural number.

Also there are “least” elements, which are comparable with all others, and have a zero property (maximum or minimum for all elements). E.g. oo . In case of it the allocation operation is terminated and only this value is returned.

Assumption:

- if $A > B > C$ then $A > C$
- if $A == B$ then B can be removed

Members**root**

`sympy.functions.elementary.miscellaneous.root(x, n, k)` → Returns the k -th n -th principal root ($k=0$).
root of x , defaulting to the

The parameter `evaluate` determines if the expression should be evaluated. If `None`, its value is taken from `global_evaluate`.

See also:

`sympy.polys.rootoftools.rootof` (page 869), `sympy.core.power.integer_nthroot` (page 155), `sqrt` (page 456), `real_root` (page 458)

References

- http://en.wikipedia.org/wiki/Square_root
- http://en.wikipedia.org/wiki/Real_root
- http://en.wikipedia.org/wiki/Root_of_unity
- http://en.wikipedia.org/wiki/Principal_value
- <http://mathworld.wolfram.com/CubeRoot.html>

Examples

```
>>> from sympy import root, Rational
>>> from sympy.abc import x, n
```

```
>>> root(x, 2)
sqrt(x)
```

```
>>> root(x, 3)
x**(1/3)
```

```
>>> root(x, n)
x**(1/n)
```

```
>>> root(x, -Rational(2, 3))
x**(-3/2)
```

To get the k-th n-th root, specify k:

```
>>> root(-2, 3, 2)
- (-1)**(2/3)*2**(1/3)
```

To get all n n-th roots you can use the rootof function. The following examples show the roots of unity for n equal 2, 3 and 4:

```
>>> from sympy import rootof, I
```

```
>>> [rootof(x**2 - 1, i) for i in range(2)]
[-1, 1]
```

```
>>> [rootof(x**3 - 1, i) for i in range(3)]
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

```
>>> [rootof(x**4 - 1, i) for i in range(4)]
[-1, 1, -I, I]
```

SymPy, like other symbolic algebra systems, returns the complex root of negative numbers. This is the principal root and differs from the text-book result that one might be expecting. For example, the cube root of -8 does not come back as -2:

```
>>> root(-8, 3)
2*(-1)**(1/3)
```

The `real_root` function can be used to either make the principal result real (or simply to return the real root directly):

```
>>> from sympy import real_root
>>> real_root(_)
-2
>>> real_root(-32, 5)
-2
```

Alternatively, the $n/2$ -th n-th root of a negative number can be computed with `root`:

```
>>> root(-32, 5, 5//2)
-2
```

sqrt

`sympy.functions.elementary.miscellaneous.sqrt`(arg, evaluate=None)

The square root function

`sqrt(x)` -> Returns the principal square root of x.

The parameter `evaluate` determines if the expression should be evaluated. If `None`, its value is taken from `global_evaluate`

See also:

[sympy.polys.rootoftools.rootof](#) (page 869), [root](#) (page 454), [real_root](#) (page 458)

References

[R174] (page 1859), [R175] (page 1859)

Examples

```
>>> from sympy import sqrt, Symbol
>>> x = Symbol('x')
```

```
>>> sqrt(x)
sqrt(x)
```

```
>>> sqrt(x)**2
x
```

Note that `sqrt(x**2)` does not simplify to `x`.

```
>>> sqrt(x**2)
sqrt(x**2)
```

This is because the two are not equal to each other in general. For example, consider `x == -1`:

```
>>> from sympy import Eq
>>> Eq(sqrt(x**2), x).subs(x, -1)
False
```

This is because `sqrt` computes the principal square root, so the square may put the argument in a different branch. This identity does hold if `x` is positive:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the `powdenest()` function with the `force` option set to `True`:

```
>>> from sympy import powdenest
>>> sqrt(x**2)
sqrt(x**2)
>>> powdenest(sqrt(x**2), force=True)
x
```

To get both branches of the square root you can use the `rootof` function:

```
>>> from sympy import rootof
```

```
>>> [rootof(x**2-3,i) for i in (0,1)]
[-sqrt(3), sqrt(3)]
```

cbrt

`sympy.functions.elementary.miscellaneous.cbrt(arg, evaluate=None)`

This function computes the principal cube root of *arg*, so it's just a shortcut for *arg* * *Rational*(1,3).

The parameter `evaluate` determines if the expression should be evaluated. If `None`, its value is taken from `global_evaluate`.

See also:

`sympy.polys.rootoftools.rootof` (page 869), `root` (page 454), `real_root` (page 458)

References

- http://en.wikipedia.org/wiki/Cube_root
- http://en.wikipedia.org/wiki/Principal_value

Examples

```
>>> from sympy import cbrt, Symbol
>>> x = Symbol('x')
```

```
>>> cbrt(x)
x**(1/3)
```

```
>>> cbrt(x)**3
x
```

Note that `cbrt(x**3)` does not simplify to `x`.

```
>>> cbrt(x**3)
(x**3)**(1/3)
```

This is because the two are not equal to each other in general. For example, consider $x = -1$:

```
>>> from sympy import Eq
>>> Eq(cbrt(x**3), x).subs(x, -1)
False
```

This is because `cbrt` computes the principal cube root, this identity does hold if x is positive:

```
>>> y = Symbol('y', positive=True)
>>> cbrt(y**3)
y
```

real_root

`sympy.functions.elementary.miscellaneous.real_root`(arg, n=None, evaluate=None)

Return the real n th-root of `arg` if possible. If n is omitted then all instances of $(-n)^{(1/\text{odd})}$ will be changed to $-n^{(1/\text{odd})}$; this will only create a real root of a principal root – the presence of other factors may cause the result to not be real.

The parameter `evaluate` determines if the expression should be evaluated. If `None`, its value is taken from `global_evaluate`.

See also:

[sympy.polys.rootoftools.rootof](#) (page 869), [sympy.core.power.integer_nthroot](#) (page 155), [root](#) (page 454), [sqrt](#) (page 456)

Examples

```
>>> from sympy import root, real_root, Rational
>>> from sympy.abc import x, n
```

```
>>> real_root(-8, 3)
-2
>>> root(-8, 3)
2*(-1)**(1/3)
>>> real_root(_)
-2
```

If one creates a non-principal root and applies `real_root`, the result will not be real (so use with caution):

```
>>> root(-8, 3, 2)
-2*(-1)**(2/3)
>>> real_root(_)
-2*(-1)**(2/3)
```

Combinatorial

This module implements various combinatorial functions.

bell

class `sympy.functions.combinatorial.numbers.bell`

Bell numbers / Bell polynomials

The Bell numbers satisfy $B_0 = 1$ and

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$

They are also given by:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

The Bell polynomials are given by $B_0(x) = 1$ and

$$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$

The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots, x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_3+\dots=k \\ j_1+2j_2+3j_3+\dots=n}} \frac{n!}{j_1!j_2!\dots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \dots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- `bell(n)` gives the n^{th} Bell number, B_n .
- `bell(n, x)` gives the n^{th} Bell polynomial, $B_n(x)$.
- `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$.

See also:

[bernoulli](#) (page 460), [catalan](#) (page 462), [euler](#) (page 464), [fibonacci](#) (page 468), [harmonic](#) (page 469), [lucas](#) (page 471), [partition](#)

Notes

Not to be confused with Bernoulli numbers and Bernoulli polynomials, which use the same notation.

References

[R85] (page 1859), [R86] (page 1859), [R87] (page 1859)

Examples

```
>>> from sympy import bell, Symbol, symbols
```

```
>>> [bell(n) for n in range(11)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
>>> bell(30)
846749014511809332450147
>>> bell(4, Symbol('t'))
t**4 + 6*t**3 + 7*t**2 + t
>>> bell(6, 2, symbols('x:6')[1:])
6*x1*x5 + 15*x2*x4 + 10*x3**2
```

bernoulli

class sympy.functions.combinatorial.numbers.bernoulli

Bernoulli numbers / Bernoulli polynomials

The Bernoulli numbers are a sequence of rational numbers defined by $B_0 = 1$ and the recursive relation ($n > 0$):

$$0 = \sum_{k=0}^n \binom{n}{k} B_k = 0$$

They are also commonly defined by their exponential generating function, which is $x/(\exp(x) - 1)$. For odd indices > 1 , the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \sum_{k=0}^n \binom{n}{k} B_k x^{n-k}$$

Bernoulli numbers and Bernoulli polynomials are related as $B_n(0) = B_n$.

We compute Bernoulli numbers using Ramanujan's formula:

$$B_n = (A(n) - S(n)) / \sum_{k=0}^{n-3} \binom{n-1}{k} B_k$$

where $A(n) = (n+3)/3$ when $n = 0$ or $2 \pmod{6}$, $A(n) = -(n+3)/6$ when $n = 4 \pmod{6}$, and:

$$S(n) = \sum_{k=1}^{\lfloor n/6 \rfloor} \binom{n-3}{6k-3} B_{6k-3}$$

This formula is similar to the sum given in the definition, but cuts 2/3 of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the n th Bernoulli number, B_n

- `bernoulli(n, x)` gives the n th Bernoulli polynomial in x , $B_n(x)$

See also:

[bell](#) (page 459), [catalan](#) (page 462), [euler](#) (page 464), [fibonacci](#) (page 468), [harmonic](#) (page 469), [lucas](#) (page 471), [partition](#)

References

[R88] (page 1859), [R89] (page 1859), [R90] (page 1859), [R91] (page 1859)

Examples

```
>>> from sympy import bernoulli

>>> [bernoulli(n) for n in range(11)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66]
>>> bernoulli(1000001)
0
```

binomial**class sympy.functions.combinatorial.factorials.binomial**

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

$$C(n, k) = n! / (k!(n-k)!) \text{ or } C(n, k) = \frac{n!}{k!}$$

First, in a strict combinatorial sense it defines the number of ways we can choose ' k ' elements from a set of ' n ' elements. In this case both arguments are nonnegative integers and binomial is computed using an efficient algorithm based on prime factorization.

The other definition is generalization for arbitrary ' n ', however ' k ' must also be nonnegative. This case is very useful when evaluating summations.

For the sake of convenience for negative integer ' k ' this function will return zero no matter what valued is the other argument.

To expand the binomial when n is a symbol, use either `expand_func()` or `expand(func=True)`. The former will keep the polynomial in factored form while the latter will expand the polynomial itself. See examples for details.

References

[R92] (page 1859)

Examples

```
>>> from sympy import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True, positive=True)
```

```
>>> binomial(15, 8)
6435
```

```
>>> binomial(n, -1)
0
```

Rows of Pascal's triangle can be generated with the binomial function:

```
>>> for N in range(8):
...     print([ binomial(N, i) for i in range(N + 1)])
...
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

As can a given diagonal, e.g. the 4th diagonal:

```
>>> N = -4
>>> [ binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]
```

```
>>> binomial(Rational(5, 4), 3)
-5/128
>>> binomial(Rational(-5, 4), 3)
-195/128
```

```
>>> binomial(n, 3)
binomial(n, 3)
```

```
>>> binomial(n, 3).expand(func=True)
n**3/6 - n**2/2 + n/3
```

```
>>> expand_func(binomial(n, 3))
n*(n - 2)*(n - 1)/6
```

catalan

class sympy.functions.combinatorial.numbers.catalan

Catalan numbers

The n-th catalan number is given by:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

- catalan(n) gives the n-th Catalan number, C_n

See also:

[bell](#) (page 459), [bernoulli](#) (page 460), [euler](#) (page 464), [fibonacci](#) (page 468), [harmonic](#) (page 469), [lucas](#) (page 471), [partition](#), [sympy.functions.combinatorial.factorials.binomial](#) (page 461)

References

[R93] (page 1859), [R94] (page 1859), [R95] (page 1859), [R96] (page 1859)

Examples

```
>>> from sympy import (Symbol, binomial, gamma, hyper, polygamma,
...                     catalan, diff, combsimp, Rational, I)
```

```
>>> [ catalan(i) for i in range(1,10) ]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> n = Symbol("n", integer=True)
```

```
>>> catalan(n)
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```
>>> catalan(n).rewrite(binomial)
binomial(2*n, n)/(n + 1)
```

```
>>> catalan(n).rewrite(gamma)
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))
```

```
>>> catalan(n).rewrite(hyper)
hyper((-n + 1, -n), (2,), 1)
```

For some non-integer values of n we can get closed form expressions by rewriting in terms of gamma functions:

```
>>> catalan(Rational(1,2)).rewrite(gamma)
8/(3*pi)
```

We can differentiate the Catalan numbers $C(n)$ interpreted as a continuous real function in n :

```
>>> diff(catalan(n), n)
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```

As a more advanced example consider the following ratio between consecutive numbers:

```
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```
>>> catalan(I).rewrite(gamma)
4*I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```
>>> catalan(I).evalf(20)
0.39764993382373624267 - 0.020884341620842555705*I
```

euler

class sympy.functions.combinatorial.numbers.euler

Euler numbers / Euler polynomials

The Euler numbers are given by:

$$E_{2n} = \sum_{k=1}^{2n+1} \sum_{j=0}^k \frac{(-1)^j}{2^k} \frac{k!}{j!} \frac{E_{2n+1-k}}{2^k} \quad E_{2n+1} = 0$$

Euler numbers and Euler polynomials are related by

$$E_n = 2^n E_n\left(\frac{1}{2}\right).$$

We compute symbolic Euler polynomials using [5]

$$E_n(x) = \sum_{k=0}^n \binom{n}{k} \frac{E_k}{2^k} \left(x - \frac{1}{2}\right)^{n-k}.$$

However, numerical evaluation of the Euler polynomial is computed more efficiently (and more accurately) using the mpmath library.

- `euler(n)` gives the n -th Euler number, E_n .
- `euler(n, x)` gives the n -th Euler polynomial, $E_n(x)$.

See also:

[bell](#) (page 459), [bernoulli](#) (page 460), [catalan](#) (page 462), [fibonacci](#) (page 468), [harmonic](#) (page 469), [lucas](#) (page 471), [partition](#)

References

[R97] (page 1859), [R98] (page 1859), [R99] (page 1859), [R100] (page 1859), [R101] (page 1859)

Examples

```
>>> from sympy import Symbol, S
>>> from sympy.functions import euler
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> n = Symbol("n")
>>> euler(n+2*n)
euler(3*n)
```

```
>>> x = Symbol("x")
>>> euler(n, x)
euler(n, x)
```

```
>>> euler(0, x)
1
>>> euler(1, x)
x - 1/2
>>> euler(2, x)
x**2 - x
>>> euler(3, x)
x**3 - 3*x**2/2 + 1/4
>>> euler(4, x)
x**4 - 2*x**3 + x
```

```
>>> euler(12, S.Half)
2702765/4096
>>> euler(12)
2702765
```

factorial

class sympy.functions.combinatorial.factorials.**factorial**

Implementation of factorial function over nonnegative integers. By convention (consistent with the gamma function and the binomial coefficients), factorial of a negative integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in which n objects can be permuted. It also arises in calculus, probability, number theory, etc.

There is strict relation of factorial with gamma function. In fact $n! = \text{gamma}(n+1)$ for nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simplification.

Computation of the factorial is done using two algorithms. For small arguments a pre-computed look up table is used. However for bigger input algorithm Prime-Swing is used. It is the fastest algorithm known and computes $n!$ via prime factorization of special class of numbers, called here the 'Swing Numbers'.

See also:

[factorial2](#) (page 467), [RisingFactorial](#) (page 472), [FallingFactorial](#) (page 468)

Examples

```
>>> from sympy import Symbol, factorial, S
>>> n = Symbol('n', integer=True)
```

```
>>> factorial(0)
1
```

```
>>> factorial(7)
5040
```

```
>>> factorial(-2)
zoo
```

```
>>> factorial(n)
factorial(n)
```

```
>>> factorial(2*n)
factorial(2*n)
```

```
>>> factorial(S(1)/2)
factorial(1/2)
```

subfactorial

class `sympy.functions.combinatorial.factorials.subfactorial`

The subfactorial counts the derangements of n items and is defined for non-negative integers as:

$$!n = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{for } n = 1 \\ (n - 1) * (!n - 1 + !n - 2) & \text{for } n > 1 \end{cases}$$

It can also be written as $\text{int}(\text{round}(n!/\exp(1)))$ but the recursive definition with caching is implemented for this function.

An interesting analytic expression is the following [\[R103\]](#) (page 1859)

$$!x = \Gamma(x + 1, -1)/e$$

which is valid for non-negative integers x . The above formula is not very useful incase of non-integers. $\Gamma(x + 1, -1)$ is single-valued only for integral arguments x , elsewhere on the positive real axis it has an infinite number of branches none of which are real.

See also:

[sympy.functions.combinatorial.factorials.factorial](#) (page 465), [sympy.utilities.iterables.generate_derangements](#) (page 1420), [sympy.functions.special.gamma_functions.uppergamma](#) (page 488)

References

[R102] (page 1859), [R103] (page 1859)

Examples

```
>>> from sympy import subfactorial
>>> from sympy.abc import n
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

factorial2 / double factorial

class sympy.functions.combinatorial.factorials.**factorial2**

The double factorial $n!!$, not to be confused with $(n)!$

The double factorial is defined for nonnegative integers and for odd negative integers as:

$$n!! = \begin{cases} n!(n-2)(n-4)\dots*1 & \text{for } n \text{ positive odd} \\ n!(n-2)(n-4)\dots*2 & \text{for } n \text{ positive even} \\ 1 & \text{for } n = 0 \\ (n+2)!! / (n+2) & \text{for } n \text{ negative odd} \end{cases}$$

See also:

[factorial](#) (page 465), [RisingFactorial](#) (page 472), [FallingFactorial](#) (page 468)

References

[R104] (page 1859)

Examples

```
>>> from sympy import factorial2, var
>>> var('n')
n
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
>>> factorial2(-5)
1/3
```

FallingFactorial

`class sympy.functions.combinatorial.factorials.FallingFactorial`

Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by

$$\text{ff}(x, k) = x * (x-1) * \dots * (x - k+1)$$

where 'x' can be arbitrary expression and 'k' is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or visit <http://mathworld.wolfram.com/FallingFactorial.html> page.

When x is a Poly instance of degree ≥ 1 with single variable, $\text{ff}(x, k) = x(y) * x(y-1) * \dots * x(y-k+1)$, where y is the variable of x. This is as described in Peter Paule, "Greatest Factorial Factorization and Symbolic Summation", Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.

```
>>> from sympy import ff, factorial, rf, gamma, polygamma, binomial, symbols, Poly
>>> from sympy.abc import x, k
>>> n, m = symbols('n m', integer=True)
>>> ff(x, 0)
1
>>> ff(5, 5)
120
>>> ff(x, 5) == x*(x-1)*(x-2)*(x-3)*(x-4)
True
>>> ff(Poly(x**2, x), 2)
Poly(x**4 - 2*x**3 + x**2, x, domain='ZZ')
>>> ff(n, n)
factorial(n)
```

Rewrite

```
>>> ff(x, k).rewrite(gamma)
(-1)**k*gamma(k - x)/gamma(-x)
>>> ff(x, k).rewrite(rf)
RisingFactorial(-k + x + 1, k)
>>> ff(x, m).rewrite(binomial)
binomial(x, m)*factorial(m)
>>> ff(n, m).rewrite(factorial)
factorial(n)/factorial(-m + n)
```

See also:

[factorial](#) (page 465), [factorial2](#) (page 467), [RisingFactorial](#) (page 472)

References

[R105] (page 1859)

fibonacci

`class sympy.functions.combinatorial.numbers.fibonacci`

Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms $F_0 = 0$, $F_1 = 1$ and the two-term recurrence relation $F_n = F_{n-1} + F_{n-2}$. This definition

extended to arbitrary real and complex arguments using the formula

$$F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

The Fibonacci polynomials are defined by $F_1(x) = 1$, $F_2(x) = x$, and $F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$ for $n > 2$. For all positive integers n , $F_n(1) = F_n$.

- `fibonacci(n)` gives the n th Fibonacci number, F_n
- `fibonacci(n, x)` gives the n th Fibonacci polynomial in x , $F_n(x)$

See also:

[bell](#) (page 459), [bernoulli](#) (page 460), [catalan](#) (page 462), [euler](#) (page 464), [harmonic](#) (page 469), [lucas](#) (page 471), [partition](#)

References

[\[R106\]](#) (page 1859), [\[R107\]](#) (page 1859)

Examples

```
>>> from sympy import fibonacci, Symbol
```

```
>>> [fibonacci(x) for x in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci(5, Symbol('t'))
t**4 + 3*t**2 + 1
```

harmonic

class `sympy.functions.combinatorial.numbers.harmonic`

Harmonic numbers

The n th harmonic number is given by $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$.

More generally:

$$H_{n,m} = \sum_{k=1}^n \frac{1}{k^m}$$

As $n \rightarrow \infty$, $H_{n,m} \rightarrow \zeta(m)$, the Riemann zeta function.

- `harmonic(n)` gives the n th harmonic number, H_n
- `harmonic(n, m)` gives the n th generalized harmonic number of order m , $H_{n,m}$, where `harmonic(n) == harmonic(n, 1)`

See also:

[bell](#) (page 459), [bernoulli](#) (page 460), [catalan](#) (page 462), [euler](#) (page 464), [fibonacci](#) (page 468), [lucas](#) (page 471), [partition](#)

References

[R108] (page 1859), [R109] (page 1859), [R110] (page 1859)

Examples

```
>>> from sympy import harmonic, oo
```

```
>>> [harmonic(n) for n in range(6)]
[0, 1, 3/2, 11/6, 25/12, 137/60]
>>> [harmonic(n, 2) for n in range(6)]
[0, 1, 5/4, 49/36, 205/144, 5269/3600]
>>> harmonic(oo, 2)
pi**2/6
```

```
>>> from sympy import Symbol, Sum
>>> n = Symbol("n")
```

```
>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))
```

We can evaluate harmonic numbers for all integral and positive rational arguments:

```
>>> from sympy import S, expand_func, simplify
>>> harmonic(8)
761/280
>>> harmonic(11)
83711/27720
```

```
>>> H = harmonic(1/S(3))
>>> H
harmonic(1/3)
>>> He = expand_func(H)
>>> He
-log(6) - sqrt(3)*pi/6 + 2*Sum(log(sin(_k*pi/3))*cos(2*_k*pi/3), (_k, 1, 1))
+ 3*Sum(1/(3*_k + 1), (_k, 0, 0))
>>> He.doit()
-log(6) - sqrt(3)*pi/6 - log(sqrt(3)/2) + 3
>>> H = harmonic(25/S(7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(pi/7)**(-2*cos(pi/7))*sin(2*pi/7)**(2*cos(16*pi/7))*cos(pi/14)**(-
↪ 2*sin(pi/14))/14)
+ pi*tan(pi/14)/2 + 30247/9900
>>> He.n(40)
1.983697455232980674869851942390639915940
>>> harmonic(25/S(7)).n(40)
1.983697455232980674869851942390639915940
```

We can rewrite harmonic numbers in terms of polygamma functions:

```
>>> from sympy import digamma, polygamma
>>> m = Symbol("m")
```

```
>>> harmonic(n).rewrite(digamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n).rewrite(polygamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n,3).rewrite(polygamma)
polygamma(2, n + 1)/2 - polygamma(2, 1)/2
```

```
>>> harmonic(n,m).rewrite(polygamma)
(-1)**m*(polygamma(m - 1, 1) - polygamma(m - 1, n + 1))/factorial(m - 1)
```

Integer offsets in the argument can be pulled out:

```
>>> from sympy import expand_func
```

```
>>> expand_func(harmonic(n+4))
harmonic(n) + 1/(n + 4) + 1/(n + 3) + 1/(n + 2) + 1/(n + 1)
```

```
>>> expand_func(harmonic(n-4))
harmonic(n) - 1/(n - 1) - 1/(n - 2) - 1/(n - 3) - 1/n
```

Some limits can be computed as well:

```
>>> from sympy import limit, oo
```

```
>>> limit(harmonic(n), n, oo)
oo
```

```
>>> limit(harmonic(n, 2), n, oo)
pi**2/6
```

```
>>> limit(harmonic(n, 3), n, oo)
-polygamma(2, 1)/2
```

However we can not compute the general relation yet:

```
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals $\zeta(m)$ for $m > 1$.

lucas

class sympy.functions.combinatorial.numbers.lucas
Lucas numbers

Lucas numbers satisfy a recurrence relation similar to that of the Fibonacci sequence, in which each term is the sum of the preceding two. They are generated by choosing the initial values $L_0 = 2$ and $L_1 = 1$.

- `lucas(n)` gives the n th Lucas number

See also:

[bell](#) (page 459), [bernoulli](#) (page 460), [catalan](#) (page 462), [euler](#) (page 464), [fibonacci](#) (page 468), [harmonic](#) (page 469), [partition](#)

References

[R111] (page 1859), [R112] (page 1859)

Examples

```
>>> from sympy import lucas
```

```
>>> [lucas(x) for x in range(11)]  
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

MultiFactorial

`class sympy.functions.combinatorial.factorials.MultiFactorial`

RisingFactorial

`class sympy.functions.combinatorial.factorials.RisingFactorial`

Rising factorial (also called Pochhammer symbol) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by:

$$\text{rf}(x, k) = x * (x + 1) * \dots * (x + k - 1)$$

where 'x' can be arbitrary expression and 'k' is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or visit <http://mathworld.wolfram.com/RisingFactorial.html> page.

When x is a Poly instance of degree ≥ 1 with a single variable, $\text{rf}(x, k) = x(y) * x(y+1) * \dots * x(y+k-1)$, where y is the variable of x. This is as described in Peter Paule, "Greatest Factorial Factorization and Symbolic Summation", Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.

See also:

[factorial](#) (page 465), [factorial2](#) (page 467), [FallingFactorial](#) (page 468)

References

[R113] (page 1859)

Examples

```

>>> from sympy import rf, symbols, factorial, ff, binomial, Poly
>>> from sympy.abc import x
>>> n, k = symbols('n k', integer=True)
>>> rf(x, 0)
1
>>> rf(1, 5)
120
>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
True
>>> rf(Poly(x**3, x), 2)
Poly(x**6 + 3*x**5 + 3*x**4 + x**3, x, domain='ZZ')

```

Rewrite

```

>>> rf(x, k).rewrite(ff)
FallingFactorial(k + x - 1, k)
>>> rf(x, k).rewrite(binomial)
binomial(k + x - 1, k)*factorial(k)
>>> rf(n, k).rewrite(factorial)
factorial(k + n - 1)/factorial(n - 1)

```

stirling

`sympy.functions.combinatorial.numbers.stirling(n, k, d=None, kind=2, signed=False)`

Return Stirling number $S(n, k)$ of the first or second (default) kind.

The sum of all Stirling numbers of the second kind for $k = 1$ through n is `bell(n)`. The recurrence relationship for these numbers is:

$$\begin{array}{llllll} \{0\} & \{n\} & \{0\} & \{n+1\} & \{n\} & \{n\} \\ \{ \} = 1; & \{ \} = \{ \} = 0; & \{ \} & \{ \} = j * \{ \} + \{ \} & & \\ \{0\} & \{0\} & \{k\} & \{k\} & \{k\} & \{k-1\} \end{array}$$

where j is:: n for Stirling numbers of the first kind $-n$ for signed Stirling numbers of the first kind k for Stirling numbers of the second kind

The first kind of Stirling number counts the number of permutations of n distinct items that have k cycles; the second kind counts the ways in which n distinct items can be partitioned into k parts. If d is given, the “reduced Stirling number of the second kind” is returned: $S^{\{d\}}(n, k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$. (This counts the ways to partition n consecutive integers into k groups with no pairwise difference less than d . See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword `signed=True`. Using this keyword automatically sets `kind` to 1.

See also:

[sympy.utilities.iterables.multiset_partitions](#) (page 1425)

References

[R114] (page 1859), [R115] (page 1859)

Examples

```
>>> from sympy.functions.combinatorial.numbers import stirling, bell
>>> from sympy.combinatorics import Permutation
>>> from sympy.utilities.iterables import multiset_partitions, permutations
```

First kind (unsigned by default):

```
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(permutations(range(4)))
>>> [sum(Permutation(p).cycles == i for p in perms) for i in range(5)]
[0, 6, 11, 6, 1]
>>> [stirling(4, i, kind=1) for i in range(5)]
[0, 6, 11, 6, 1]
```

First kind (signed):

```
>>> [stirling(4, i, signed=True) for i in range(5)]
[0, -6, 11, -6, 1]
```

Second kind:

```
>>> [stirling(10, i) for i in range(12)]
[0, 1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1, 0]
>>> sum(_) == bell(10)
True
>>> len(list(multiset_partitions(range(4), 2))) == stirling(4, 2)
True
```

Reduced second kind:

```
>>> from sympy import subsets, oo
>>> def delta(p):
...     if len(p) == 1:
...         return oo
...     return min(abs(i[0] - i[1]) for i in subsets(p, 2))
>>> parts = multiset_partitions(range(5), 3)
>>> d = 2
>>> sum(1 for p in parts if all(delta(i) >= d for i in p))
7
>>> stirling(5, 3, 2)
7
```

Enumeration

Three functions are available. Each of them attempts to efficiently compute a given combinatorial quantity for a given set or multiset which can be entered as an integer, sequence or multiset (dictionary with elements as keys and multiplicities as values). The *k* parameter indicates the number of elements to pick (or the number of partitions to make). When *k* is *None*, the sum of the enumeration for all *k* (from 0 through the number of items represented by *n*) is returned. A replacement parameter is recognized for combinations and permutations; this indicates that any item may appear with multiplicity as high as the number of items in the original set.


```
>>> from sympy.functions.combinatorial.numbers import nC, nP, nT
>>> items = 'baby'
```

nC

Calculate the number of combinations of length k .

```
>>> [nC(items, k) for k in range(len(items) + 1)], nC(items)
([1, 3, 4, 3, 1], 12)
>>> nC('aaa', 2)
1
>>> nC('abc', 2)
3
>>> nC(3, 2)
3
```

nP

Calculate the number of permutations of length k .

```
>>> [nP(items, k) for k in range(len(items) + 1)], nP(items)
([1, 3, 7, 12, 12], 35)
>>> nC('aaa', 2)
1
>>> nC('abc', 2)
3
>>> nC(3, 2)
3
```

nT

Calculate the number of partitions that have k parts.

```
>>> [nT(items, k) for k in range(len(items) + 1)], nT(items)
([0, 1, 5, 4, 1], 11)
>>> nT('aaa', 2)
1
>>> nT('abc', 2)
3
>>> nT(3, 2)
1
```

Note that the integer for n indicates identical items for nT but indicates n different items for nC and nP .

Special

DiracDelta

class sympy.functions.special.delta_functions.**DiracDelta**
The DiracDelta function and its derivatives.

DiracDelta is not an ordinary function. It can be rigorously defined either as a distribution or as a measure.

DiracDelta only makes sense in definite integrals, and in particular, integrals of the form `Integral(f(x)*DiracDelta(x - x0), (x, a, b))`, where it equals $f(x_0)$ if $a \leq x_0 \leq b$ and 0 otherwise. Formally, DiracDelta acts in some ways like a function that is 0 everywhere except at 0, but in many ways it also does not. It can often be useful to treat DiracDelta in formal ways, building up and manipulating expressions with delta functions (which may eventually be integrated), but care must be taken to not treat it as a real function. SymPy's `oo` is similar. It only truly makes sense formally in certain contexts (such as integration limits), but SymPy allows its use everywhere, and it tries to be consistent with operations on it (like $1/\infty$), but it is easy to get into trouble and get wrong results if `oo` is treated too much like a number. Similarly, if DiracDelta is treated too much like a function, it is easy to get wrong or nonsensical results.

DiracDelta function has the following properties:

1. `diff(Heaviside(x), x) = DiracDelta(x)`
2. `integrate(DiracDelta(x - a)*f(x), (x, -oo, oo)) = f(a)` and `integrate(DiracDelta(x - a)*f(x), (x, a - e, a + e)) = f(a)`
3. `DiracDelta(x) = 0` for all $x \neq 0$
4. `DiracDelta(g(x)) = Sum_i(DiracDelta(x - x_i)/abs(g'(x_i)))` Where x_i -s are the roots of g
5. `DiracDelta(-x) = DiracDelta(x)`

Derivatives of k -th order of DiracDelta have the following property:

6. `DiracDelta(x, k) = 0`, for all $x \neq 0$
7. `DiracDelta(-x, k) = -DiracDelta(x, k)` for odd k
8. `DiracDelta(-x, k) = DiracDelta(x, k)` for even k

See also:

[Heaviside](#) (page 479), [simplify](#), [is_simple](#) (page 478), [sympy.functions.special.tensor_functions.KroneckerDelta](#) (page 553)

References

[R176] (page 1859)

Examples

```
>>> from sympy import DiracDelta, diff, pi, Piecewise
>>> from sympy.abc import x, y
```

```
>>> DiracDelta(x)
DiracDelta(x)
>>> DiracDelta(1)
0
>>> DiracDelta(-1)
0
>>> DiracDelta(pi)
0
```

(continues on next page)

(continued from previous page)

```

>>> DiracDelta(x - 4).subs(x, 4)
DiracDelta(0)
>>> diff(DiracDelta(x))
DiracDelta(x, 1)
>>> diff(DiracDelta(x - 1), x, 2)
DiracDelta(x - 1, 2)
>>> diff(DiracDelta(x**2 - 1), x, 2)
2*(2*x**2*DiracDelta(x**2 - 1, 2) + DiracDelta(x**2 - 1, 1))
>>> DiracDelta(3*x).is_simple(x)
True
>>> DiracDelta(x**2).is_simple(x)
False
>>> DiracDelta((x**2 - 1)*y).expand(diracdelta=True, wrt=x)
DiracDelta(x - 1)/(2*Abs(y)) + DiracDelta(x + 1)/(2*Abs(y))

```

classmethod eval(arg, k=0)

Returns a simplified form or a value of DiracDelta depending on the argument passed by the DiracDelta object.

The eval() method is automatically called when the DiracDelta class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, eval() method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```

>>> from sympy import DiracDelta, S, Subs
>>> from sympy.abc import x

```

```

>>> DiracDelta(x)
DiracDelta(x)

```

```

>>> DiracDelta(-x, 1)
-DiracDelta(x, 1)

```

```

>>> DiracDelta(1)
0

```

```

>>> DiracDelta(5, 1)
0

```

```

>>> DiracDelta(0)
DiracDelta(0)

```

```

>>> DiracDelta(-1)
0

```

```

>>> DiracDelta(S.NaN)
nan

```

```
>>> DiracDelta(x).eval(1)
0
```

```
>>> DiracDelta(x - 100).subs(x, 5)
0
```

```
>>> DiracDelta(x - 100).subs(x, 100)
DiracDelta(0)
```

fdiff(argindex=1)

Returns the first derivative of a DiracDelta Function.

The difference between `diff()` and `fdiff()` is:- `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is just a convenience method available in the Function class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

Examples

```
>>> from sympy import DiracDelta, diff
>>> from sympy.abc import x
```

```
>>> DiracDelta(x).fdiff()
DiracDelta(x, 1)
```

```
>>> DiracDelta(x, 1).fdiff()
DiracDelta(x, 2)
```

```
>>> DiracDelta(x**2 - 1).fdiff()
DiracDelta(x**2 - 1, 1)
```

```
>>> diff(DiracDelta(x, 1)).fdiff()
DiracDelta(x, 3)
```

is_simple(self, x)

Tells whether the argument(`args[0]`) of `DiracDelta` is a linear expression in `x`.

`x` can be:

- a symbol

See also:

`simplify`, `Diracdelta`

Examples

```
>>> from sympy import DiracDelta, cos
>>> from sympy.abc import x, y
```

```
>>> DiracDelta(x*y).is_simple(x)
True
>>> DiracDelta(x*y).is_simple(y)
True
```

```
>>> DiracDelta(x**2 + x - 2).is_simple(x)
False
```

```
>>> DiracDelta(cos(x)).is_simple(x)
False
```

Heaviside

class `sympy.functions.special.delta_functions.Heaviside`

Heaviside Piecewise function

Heaviside function has the following properties [\[R177\]](#) (page 1860):

1. `diff(Heaviside(x), x) = DiracDelta(x)` (0, if $x < 0$
2. `Heaviside(x) = < ( undefined if  $x=0$  [1] ( 1, if  $x > 0$`
3. `Max(0, x).diff(x) = Heaviside(x)`

To specify the value of Heaviside at $x=0$, a second argument can be given. Omit this 2nd argument or pass None to recover the default behavior.

```
>>> from sympy import Heaviside, S
>>> from sympy.abc import x
>>> Heaviside(9)
1
>>> Heaviside(-9)
0
>>> Heaviside(0)
Heaviside(0)
>>> Heaviside(0, S.Half)
1/2
>>> (Heaviside(x) + 1).replace(Heaviside(x), Heaviside(x, 1))
Heaviside(x, 1) + 1
```

See also:

[DiracDelta](#) (page 475)

References

[\[R178\]](#) (page 1860), [\[R179\]](#) (page 1860)

classmethod `eval(arg, H0=None)`

Returns a simplified form or a value of Heaviside depending on the argument passed by the Heaviside object.

The `eval()` method is automatically called when the Heaviside class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import Heaviside, S
>>> from sympy.abc import x
```

```
>>> Heaviside(x)
Heaviside(x)
```

```
>>> Heaviside(19)
1
```

```
>>> Heaviside(0)
Heaviside(0)
```

```
>>> Heaviside(0, 1)
1
```

```
>>> Heaviside(-5)
0
```

```
>>> Heaviside(S.NaN)
nan
```

```
>>> Heaviside(x).eval(100)
1
```

```
>>> Heaviside(x - 100).subs(x, 5)
0
```

```
>>> Heaviside(x - 100).subs(x, 105)
1
```

fdiff(argindex=1)

Returns the first derivative of a Heaviside Function.

Examples

```
>>> from sympy import Heaviside, diff
>>> from sympy.abc import x
```

```
>>> Heaviside(x).fdiff()
DiracDelta(x)
```

```
>>> Heaviside(x**2 - 1).fdiff()
DiracDelta(x**2 - 1)
```

```
>>> diff(Heaviside(x)).fdiff()
DiracDelta(x, 1)
```

Singularity Function

`class sympy.functions.special.singularity_functions.SingularityFunction`

The Singularity functions are a class of discontinuous functions. They take a variable, an offset and an exponent as arguments. These functions are represented using Macaulay brackets as :

$$\text{SingularityFunction}(x, a, n) := \langle x - a \rangle^n$$

The singularity function will automatically evaluate to $\text{Derivative}(\text{DiracDelta}(x - a), x, -n - 1)$ if $n < 0$ and $(x - a)^n \text{Heaviside}(x - a)$ if $n \geq 0$.

See also:

`DiracDelta`, `Heaviside`

Examples

```
>>> from sympy import SingularityFunction, diff, Piecewise, DiracDelta, Heaviside,
↳ Symbol
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> SingularityFunction(y, -10, n)
(y + 10)**n
>>> y = Symbol('y', negative=True)
>>> SingularityFunction(y, 10, n)
0
>>> SingularityFunction(x, 4, -1).subs(x, 4)
oo
>>> SingularityFunction(x, 10, -2).subs(x, 10)
oo
>>> SingularityFunction(4, 1, 5)
243
>>> diff(SingularityFunction(x, 1, 5) + SingularityFunction(x, 1, 4), x)
4*SingularityFunction(x, 1, 3) + 5*SingularityFunction(x, 1, 4)
>>> diff(SingularityFunction(x, 4, 0), x, 2)
SingularityFunction(x, 4, -2)
>>> SingularityFunction(x, 4, 5).rewrite(Piecewise)
Piecewise(((x - 4)**5, x - 4 > 0), (0, True))
>>> expr = SingularityFunction(x, a, n)
>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> expr.subs({x: y, a: -10, n: n})
(y + 10)**n
```

The methods `rewrite(DiracDelta)`, `rewrite(Heaviside)` and `rewrite('HeavisideDiracDelta')` returns the same output. One can use any of these methods according to their choice.

```
>>> expr = SingularityFunction(x, 4, 5) + SingularityFunction(x, -3, -1) -
↳ SingularityFunction(x, 0, -2)
>>> expr.rewrite(Heaviside)
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
>>> expr.rewrite(DiracDelta)
```

(continues on next page)

(continued from previous page)

```
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
>>> expr.rewrite('HeavisideDiracDelta')
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
```

Reference

classmethod `eval(variable, offset, exponent)`

Returns a simplified form or a value of Singularity Function depending on the argument passed by the object.

The `eval()` method is automatically called when the `SingularityFunction` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import SingularityFunction, Symbol, nan
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> SingularityFunction(5, 3, 2)
4
>>> SingularityFunction(x, a, nan)
nan
>>> SingularityFunction(x, 3, 0).subs(x, 3)
1
>>> SingularityFunction(x, a, n).eval(3, 5, 1)
0
>>> SingularityFunction(x, a, n).eval(4, 1, 5)
243
>>> x = Symbol('x', positive = True)
>>> a = Symbol('a', negative = True)
>>> n = Symbol('n', nonnegative = True)
>>> SingularityFunction(x, a, n)
(-a + x)**n
>>> x = Symbol('x', negative = True)
>>> a = Symbol('a', positive = True)
>>> SingularityFunction(x, a, n)
0
```

fdiff(argindex=1)

Returns the first derivative of a DiracDelta Function.

The difference between `diff()` and `fdiff()` is:- `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is just a convenience method available in the `Function` class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

Gamma, Beta and related Functions

class `sympy.functions.special.gamma_functions.gamma`
 The gamma function

$$\Gamma(x) := \int_0^{\infty} t^{x-1} e^{-t} dt.$$

The gamma function implements the function which passes through the values of the factorial function, i.e. $\Gamma(n) = (n-1)!$ when n is an integer. More general, $\Gamma(z)$ is defined in the whole complex plane except at the negative integers where there are simple poles.

See also:

lowergamma (page 489) Lower incomplete gamma function.

uppergamma (page 488) Upper incomplete gamma function.

polygamma (page 486) Polygamma function.

loggamma (page 484) Log Gamma function.

digamma (page 487) Digamma function.

trigamma (page 488) Trigamma function.

`sympy.functions.special.beta_functions.beta` (page 490) Euler Beta function.

References

[R181] (page 1860), [R182] (page 1860), [R183] (page 1860), [R184] (page 1860)

Examples

```
>>> from sympy import S, I, pi, oo, gamma
>>> from sympy.abc import x
```

Several special values are known:

```
>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(S(3)/2)
sqrt(pi)/2
```

The Gamma function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-EulerGamma*pi**2/12 +
↳ polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.288037795340032417959588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

class `sympy.functions.special.gamma_functions.loggamma`

The loggamma function implements the logarithm of the gamma function i.e, $\log \Gamma(x)$.

See also:

gamma ([page 483](#)) Gamma function.

lowergamma ([page 489](#)) Lower incomplete gamma function.

uppergamma ([page 488](#)) Upper incomplete gamma function.

polygamma ([page 486](#)) Polygamma function.

digamma ([page 487](#)) Digamma function.

trigamma ([page 488](#)) Trigamma function.

`sympy.functions.special.beta_functions.beta` ([page 490](#)) Euler Beta function.

References

[R185] ([page 1860](#)), [R186] ([page 1860](#)), [R187] ([page 1860](#)), [R188] ([page 1860](#))

Examples

Several special values are known. For numerical integral arguments we have:

```
>>> from sympy import loggamma
>>> loggamma(-2)
oo
>>> loggamma(0)
oo
>>> loggamma(1)
0
>>> loggamma(2)
0
>>> loggamma(3)
log(2)
```

and for symbolic values:

```
>>> from sympy import Symbol
>>> n = Symbol("n", integer=True, positive=True)
>>> loggamma(n)
log(gamma(n))
>>> loggamma(-n)
oo
```

for half-integral values:

```
>>> from sympy import S, pi
>>> loggamma(S(5)/2)
log(3*sqrt(pi)/4)
>>> loggamma(n/2)
log(2**(-n + 1)*sqrt(pi)*gamma(n)/gamma(n/2 + 1/2))
```

and general rational arguments:

```
>>> from sympy import expand_func
>>> L = loggamma(S(16)/3)
>>> expand_func(L).doit()
-5*log(3) + loggamma(1/3) + log(4) + log(7) + log(10) + log(13)
>>> L = loggamma(S(19)/4)
>>> expand_func(L).doit()
-4*log(4) + loggamma(3/4) + log(3) + log(7) + log(11) + log(15)
>>> L = loggamma(S(23)/7)
>>> expand_func(L).doit()
-3*log(7) + log(2) + loggamma(2/7) + log(9) + log(16)
```

The loggamma function has the following limits towards infinity:

```
>>> from sympy import oo
>>> loggamma(oo)
oo
>>> loggamma(-oo)
zoo
```

The loggamma function obeys the mirror symmetry if $x \in \mathbb{C} \setminus \{-\infty, 0\}$:

```
>>> from sympy.abc import x
>>> from sympy import conjugate
>>> conjugate(loggamma(x))
loggamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(loggamma(x), x, 0, 4)
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + O(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from sympy import I
>>> loggamma(5).evalf(30)
3.17805383034794561964694160130
>>> loggamma(I).evalf(20)
-0.65092319930185633889 - 1.8724366472624298171*I
```

class sympy.functions.special.gamma_functions.polygamma

The function `polygamma(n, z)` returns $\log(\Gamma(z)).\text{diff}(n + 1)$.

It is a meromorphic function on $\mathbb{C}$ and defined as the $(n+1)$ -th derivative of the logarithm of the gamma function:

$$\psi^{(n)}(z) := \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z).$$

See also:

gamma (page 483) Gamma function.

lowergamma (page 489) Lower incomplete gamma function.

uppergamma (page 488) Upper incomplete gamma function.

loggamma (page 484) Log Gamma function.

digamma (page 487) Digamma function.

trigamma (page 488) Trigamma function.

sympy.functions.special.beta_functions.beta (page 490) Euler Beta function.

References

[R189] (page 1860), [R190] (page 1860), [R191] (page 1860), [R192] (page 1860)

Examples

Several special values are known:

```
>>> from sympy import S, polygamma
>>> polygamma(0, 1)
-EulerGamma
>>> polygamma(0, 1/S(2))
-2*log(2) - EulerGamma
>>> polygamma(0, 1/S(3))
-log(3) - sqrt(3)*pi/6 - EulerGamma - log(sqrt(3))
>>> polygamma(0, 1/S(4))
-pi/2 - log(4) - log(2) - EulerGamma
>>> polygamma(0, 2)
-EulerGamma + 1
>>> polygamma(0, 23)
-EulerGamma + 19093197/5173168
```

```
>>> from sympy import oo, I
>>> polygamma(0, oo)
oo
>>> polygamma(0, -oo)
```

(continues on next page)

(continued from previous page)

```

00
>>> polygamma(0, I*oo)
00
>>> polygamma(0, -I*oo)
00

```

Differentiation with respect to x is supported:

```

>>> from sympy import Symbol, diff
>>> x = Symbol("x")
>>> diff(polygamma(0, x), x)
polygamma(1, x)
>>> diff(polygamma(0, x), x, 2)
polygamma(2, x)
>>> diff(polygamma(0, x), x, 3)
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), x, 2)
polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
>>> diff(polygamma(2, x), x, 2)
polygamma(4, x)

```

```

>>> n = Symbol("n")
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), x, 2)
polygamma(n + 2, x)

```

We can rewrite polygamma functions in terms of harmonic numbers:

```

>>> from sympy import harmonic
>>> polygamma(0, x).rewrite(harmonic)
harmonic(x - 1) - EulerGamma
>>> polygamma(2, x).rewrite(harmonic)
2*harmonic(x - 1, 3) - 2*zeta(3)
>>> ni = Symbol("n", integer=True)
>>> polygamma(ni, x).rewrite(harmonic)
(-1)**(n + 1)*(-harmonic(x - 1, n + 1) + zeta(n + 1))*factorial(n)

```

`sympy.functions.special.gamma_functions.digamma(x)`

The digamma function is the first derivative of the loggamma function i.e,

$$\psi(x) := \frac{d}{dz} \log \Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}$$

In this case, `digamma(z) = polygamma(0, z)`.

See also:

gamma (page 483) Gamma function.

lowergamma (page 489) Lower incomplete gamma function.

uppergamma (page 488) Upper incomplete gamma function.

polygamma (page 486) Polygamma function.

loggamma (page 484) Log Gamma function.

trigamma (page 488) Trigamma function.

sympy.functions.special.beta_functions.beta (page 490) Euler Beta function.

References

[R193] (page 1860), [R194] (page 1860), [R195] (page 1860)

`sympy.functions.special.gamma_functions.trigamma(x)`

The trigamma function is the second derivative of the loggamma function i.e,

$$\psi^{(1)}(z) := \frac{d^2}{dz^2} \log \Gamma(z).$$

In this case, `trigamma(z) = polygamma(1, z)`.

See also:

gamma (page 483) Gamma function.

lowergamma (page 489) Lower incomplete gamma function.

uppergamma (page 488) Upper incomplete gamma function.

polygamma (page 486) Polygamma function.

loggamma (page 484) Log Gamma function.

digamma (page 487) Digamma function.

sympy.functions.special.beta_functions.beta (page 490) Euler Beta function.

References

[R196] (page 1860), [R197] (page 1860), [R198] (page 1860)

class `sympy.functions.special.gamma_functions.uppergamma`

The upper incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\Gamma(s, x) := \int_x^\infty t^{s-1} e^{-t} dt = \Gamma(s) - \gamma(s, x).$$

where $\gamma(s, x)$ is the lower incomplete gamma function, **lowergamma** (page 489). This can be shown to be the same as

$$\Gamma(s, x) = \Gamma(s) - \frac{x^s}{s} {}_1F_1 \left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x \right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt = x^{n-1} \Gamma(1-n, x).$$

See also:

gamma (page 483) Gamma function.

lowergamma (page 489) Lower incomplete gamma function.

polygamma (page 486) Polygamma function.

loggamma (page 484) Log Gamma function.

digamma (page 487) Digamma function.

trigamma (page 488) Trigamma function.

sympy.functions.special.beta_functions.beta (page 490) Euler Beta function.

References

[R199] (page 1860), [R200] (page 1860), [R201] (page 1860), [R202] (page 1860), [R203] (page 1860), [R204] (page 1860)

Examples

```
>>> from sympy import uppergamma, S
>>> from sympy.abc import s, x
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
2*(x**2/2 + x + 1)*exp(-x)
>>> uppergamma(-S(1)/2, x)
-2*sqrt(pi)*erfc(sqrt(x)) + 2*exp(-x)/sqrt(x)
>>> uppergamma(-2, x)
expint(3, x)/x**2
```

class sympy.functions.special.gamma_functions.lowergamma

The lower incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\gamma(s, x) := \int_0^x t^{s-1} e^{-t} dt = \Gamma(s) - \Gamma(s, x).$$

This can be shown to be the same as

$$\gamma(s, x) = \frac{x^s}{s} {}_1F_1 \left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x \right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

See also:

gamma (page 483) Gamma function.

uppergamma (page 488) Upper incomplete gamma function.

polygamma (page 486) Polygamma function.

loggamma (page 484) Log Gamma function.

digamma (page 487) Digamma function.

trigamma (page 488) Trigamma function.

sympy.functions.special.beta_functions.beta (page 490) Euler Beta function.

References

[R205] (page 1860), [R206] (page 1860), [R207] (page 1860), [R208] (page 1860), [R209] (page 1860)

Examples

```
>>> from sympy import lowergamma, S
>>> from sympy.abc import s, x
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
-2*(x**2/2 + x + 1)*exp(-x) + 2
>>> lowergamma(-S(1)/2, x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*exp(-x)/sqrt(x)
```

class `sympy.functions.special.beta_functions.beta`

The beta integral is called the Eulerian integral of the first kind by Legendre:

$$B(x, y) := \int_0^1 t^{x-1} (1-t)^{y-1} dt.$$

Beta function or Euler's first integral is closely associated with gamma function. The Beta function often used in probability theory and mathematical statistics. It satisfies properties like:

$$\begin{aligned} B(a, 1) &= \frac{1}{a} \\ B(a, b) &= B(b, a) \\ B(a, b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \end{aligned}$$

Therefore for integral values of a and b:

$$B = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

See also:

`sympy.functions.special.gamma_functions.gamma` (page 483) Gamma function.

`sympy.functions.special.gamma_functions.uppergamma` (page 488) Upper incomplete gamma function.

`sympy.functions.special.gamma_functions.lowergamma` (page 489) Lower incomplete gamma function.

`sympy.functions.special.gamma_functions.polygamma` (page 486) Polygamma function.

`sympy.functions.special.gamma_functions.loggamma` (page 484) Log Gamma function.

`sympy.functions.special.gamma_functions.digamma` (page 487) Digamma function.

`sympy.functions.special.gamma_functions.trigamma` (page 488) Trigamma function.

References

[R210] (page 1861), [R211] (page 1861), [R212] (page 1861)

Examples

```
>>> from sympy import I, pi
>>> from sympy.abc import x,y
```

The Beta function obeys the mirror symmetry:

```
>>> from sympy import beta
>>> from sympy import conjugate
>>> conjugate(beta(x,y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both x and y is supported:

```
>>> from sympy import beta
>>> from sympy import diff
>>> diff(beta(x,y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)
```

```
>>> from sympy import beta
>>> from sympy import diff
>>> diff(beta(x,y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from sympy import beta
>>> beta(pi,pi).evalf(40)
0.02671848900111377452242355235388489324562
```

```
>>> beta(1+I,1+I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

Error Functions and Fresnel Integrals

class `sympy.functions.special.error_functions.erf`

The Gauss error function. This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

See also:

erfc (page 492) Complementary error function.

erfi (page 494) Imaginary error function.

erf2 (page 495) Two-argument error function.

erfinv (page 496) Inverse error function.

erfcinv (page 497) Inverse Complementary error function.

erf2inv (page 497) Inverse two-argument error function.

References

[R213] (page 1861), [R214] (page 1861), [R215] (page 1861), [R216] (page 1861)

Examples

```
>>> from sympy import I, oo, erf
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erf(0)
0
>>> erf(oo)
1
>>> erf(-oo)
-1
>>> erf(I*oo)
oo*I
>>> erf(-I*oo)
-oo*I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erf(-z)
-erf(z)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erf(z), z)
2*exp(-z**2)/sqrt(pi)
```

We can numerically evaluate the error function to arbitrary precision on the whole complex plane:

```
>>> erf(4).evalf(30)
0.999999984582742099719981147840
```

```
>>> erf(-4*I).evalf(30)
-1296959.73071763923152794095062*I
```

class sympy.functions.special.error_functions.**erfc**
Complementary Error Function. The function is defined as:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

See also:**erf** (page 491) Gaussian error function.**erfi** (page 494) Imaginary error function.**erf2** (page 495) Two-argument error function.**erfinv** (page 496) Inverse error function.**erfcinv** (page 497) Inverse Complementary error function.**erf2inv** (page 497) Inverse two-argument error function.**References**[\[R217\]](#) (page 1861), [\[R218\]](#) (page 1861), [\[R219\]](#) (page 1861), [\[R220\]](#) (page 1861)**Examples**

```
>>> from sympy import I, oo, erfc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfc(0)
1
>>> erfc(oo)
0
>>> erfc(-oo)
2
>>> erfc(I*oo)
-oo*I
>>> erfc(-I*oo)
oo*I
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfc(z), z)
-2*exp(-z**2)/sqrt(pi)
```

It also follows

```
>>> erfc(-z)
-erfc(z) + 2
```

We can numerically evaluate the complementary error function to arbitrary precision on the whole complex plane:

```
>>> erfc(4).evalf(30)
0.0000000154172579002800188521596734869
```

```
>>> erfc(4*I).evalf(30)
1.0 - 1296959.73071763923152794095062*I
```

class `sympy.functions.special.error_functions.eri`
Imaginary error function. The function `erfi` is defined as:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt$$

See also:

erf (page 491) Gaussian error function.

erfc (page 492) Complementary error function.

erf2 (page 495) Two-argument error function.

erfinv (page 496) Inverse error function.

erfcinv (page 497) Inverse Complementary error function.

erf2inv (page 497) Inverse two-argument error function.

References

[R221] (page 1861), [R222] (page 1861), [R223] (page 1861)

Examples

```
>>> from sympy import I, oo, erfi
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfi(0)
0
>>> erfi(oo)
oo
>>> erfi(-oo)
-oo
>>> erfi(I*oo)
I
>>> erfi(-I*oo)
-I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erfi(-z)
-erfi(z)
```

```
>>> from sympy import conjugate
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfi(z), z)
2*exp(z**2)/sqrt(pi)
```

We can numerically evaluate the imaginary error function to arbitrary precision on the whole complex plane:

```
>>> erfi(2).evalf(30)
18.5648024145755525987042919132
```

```
>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

class `sympy.functions.special.error_functions.erf2`
Two-argument error function. This function is defined as:

$$\operatorname{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} dt$$

See also:

erf (page 491) Gaussian error function.

erfc (page 492) Complementary error function.

erfi (page 494) Imaginary error function.

erfinv (page 496) Inverse error function.

erfcinv (page 497) Inverse Complementary error function.

erf2inv (page 497) Inverse two-argument error function.

References

[R224] (page 1861)

Examples

```
>>> from sympy import I, oo, erf2
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2(0, 0)
0
>>> erf2(x, x)
0
>>> erf2(x, oo)
-erf(x) + 1
>>> erf2(x, -oo)
-erf(x) - 1
>>> erf2(oo, y)
erf(y) - 1
```

(continues on next page)

(continued from previous page)

```
>>> erf2(-oo, y)
erf(y) + 1
```

In general one can pull out factors of -1:

```
>>> erf2(-x, -y)
-erf2(x, y)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to x, y is supported:

```
>>> from sympy import diff
>>> diff(erf2(x, y), x)
-2*exp(-x**2)/sqrt(pi)
>>> diff(erf2(x, y), y)
2*exp(-y**2)/sqrt(pi)
```

class `sympy.functions.special.error_functions.erfinv`
Inverse Error Function. The `erfinv` function is defined as:

$$\operatorname{erf}(x) = y \quad \Rightarrow \quad \operatorname{erfinv}(y) = x$$

See also:

erf (page 491) Gaussian error function.

erfc (page 492) Complementary error function.

erfi (page 494) Imaginary error function.

erf2 (page 495) Two-argument error function.

erfcinv (page 497) Inverse Complementary error function.

erf2inv (page 497) Inverse two-argument error function.

References

[R225] (page 1861), [R226] (page 1861)

Examples

```
>>> from sympy import I, oo, erfinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfinv(0)
0
>>> erfinv(1)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfinv(x), x)
sqrt(pi)*exp(erfinv(x)**2)/2
```

We can numerically evaluate the inverse error function to arbitrary precision on $[-1, 1]$:

```
>>> erfinv(0.2).evalf(30)
0.179143454621291692285822705344
```

class `sympy.functions.special.error_functions.erfcinv`

Inverse Complementary Error Function. The `erfcinv` function is defined as:

$$\operatorname{erfc}(x) = y \quad \Rightarrow \quad \operatorname{erfcinv}(y) = x$$

See also:

erf (page 491) Gaussian error function.

erfc (page 492) Complementary error function.

erfi (page 494) Imaginary error function.

erf2 (page 495) Two-argument error function.

erfinv (page 496) Inverse error function.

erf2inv (page 497) Inverse two-argument error function.

References

[R227] (page 1861), [R228] (page 1861)

Examples

```
>>> from sympy import I, oo, erfcinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfcinv(x), x)
-sqrt(pi)*exp(erfcinv(x)**2)/2
```

class `sympy.functions.special.error_functions.erf2inv`

Two-argument Inverse error function. The `erf2inv` function is defined as:

$$\operatorname{erf2}(x, w) = y \quad \Rightarrow \quad \operatorname{erf2inv}(x, y) = w$$

See also:

erf (page 491) Gaussian error function.
erfc (page 492) Complementary error function.
erfi (page 494) Imaginary error function.
erf2 (page 495) Two-argument error function.
erfinv (page 496) Inverse error function.
erfcinv (page 497) Inverse complementary error function.

References

[R229] (page 1861)

Examples

```
>>> from sympy import I, oo, erf2inv, erfinv, erfcinv
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2inv(0, 0)
0
>>> erf2inv(1, 0)
1
>>> erf2inv(0, 1)
oo
>>> erf2inv(0, y)
erfinv(y)
>>> erf2inv(oo, y)
erfcinv(-y)
```

Differentiation with respect to x and y is supported:

```
>>> from sympy import diff
>>> diff(erf2inv(x, y), x)
exp(-x**2 + erf2inv(x, y)**2)
>>> diff(erf2inv(x, y), y)
sqrt(pi)*exp(erf2inv(x, y)**2)/2
```

class sympy.functions.special.error_functions.FresnelIntegral
Base class for the Fresnel integrals.

class sympy.functions.special.error_functions.fresnels
Fresnel integral S.

This function is defined by

$$S(z) = \int_0^z \sin \frac{\pi}{2} t^2 dt.$$

It is an entire function.

See also:

fresnelc (page 500) Fresnel cosine integral.

References

[R230] (page 1861), [R231] (page 1861), [R232] (page 1861), [R233] (page 1861), [R234] (page 1861)

Examples

```
>>> from sympy import I, oo, fresnels
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnels(0)
0
>>> fresnels(oo)
1/2
>>> fresnels(-oo)
-1/2
>>> fresnels(I*oo)
-I/2
>>> fresnels(-I*oo)
I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnels(-z)
-fresnels(z)
>>> fresnels(I*z)
-I*fresnels(z)
```

The Fresnel S integral obeys the mirror symmetry $\overline{S(z)} = S(\bar{z})$:

```
>>> from sympy import conjugate
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> from sympy import integrate, pi, sin, gamma, expand_func
>>> integrate(sin(pi*z**2/2), z)
3*fresnels(z)*gamma(3/4)/(4*gamma(7/4))
>>> expand_func(integrate(sin(pi*z**2/2), z))
fresnels(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnels(2).evalf(30)
0.343415678363698242195300815958
```

```
>>> fresnels(-2*I).evalf(30)
0.343415678363698242195300815958*I
```

class `sympy.functions.special.error_functions.fresnelc`
Fresnel integral C.

This function is defined by

$$C(z) = \int_0^z \cos \frac{\pi}{2} t^2 dt.$$

It is an entire function.

See also:

fresnels (page 498) Fresnel sine integral.

References

[R235] (page 1861), [R236] (page 1861), [R237] (page 1861), [R238] (page 1861), [R239] (page 1861)

Examples

```
>>> from sympy import I, oo, fresnelc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnelc(0)
0
>>> fresnelc(oo)
1/2
>>> fresnelc(-oo)
-1/2
>>> fresnelc(I*oo)
I/2
>>> fresnelc(-I*oo)
-I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnelc(-z)
-fresnelc(z)
>>> fresnelc(I*z)
I*fresnelc(z)
```

The Fresnel C integral obeys the mirror symmetry $\overline{C(z)} = C(\bar{z})$:

```
>>> from sympy import conjugate
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> from sympy import integrate, pi, cos, gamma, expand_func
>>> integrate(cos(pi*z**2/2), z)
fresnelc(z)*gamma(1/4)/(4*gamma(5/4))
>>> expand_func(integrate(cos(pi*z**2/2), z))
fresnelc(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnelc(2).evalf(30)
0.488253406075340754500223503357
```

```
>>> fresnelc(-2*I).evalf(30)
-0.488253406075340754500223503357*I
```

Exponential, Logarithmic and Trigonometric Integrals

class `sympy.functions.special.error_functions.Ei`

The classical exponential integral.

For use in SymPy, this function is defined as

$$\operatorname{Ei}(x) = \sum_{n=1}^{\infty} \frac{x^n}{n n!} + \log(x) + \gamma,$$

where γ is the Euler-Mascheroni constant.

If x is a polar number, this defines an analytic function on the Riemann surface of the logarithm. Otherwise this defines an analytic function in the cut plane $\mathbb{C} \setminus (-\infty, 0]$.

Background

The name exponential integral comes from the following statement:

$$\operatorname{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt$$

If the integral is interpreted as a Cauchy principal value, this statement holds for $x > 0$ and $\operatorname{Ei}(x)$ as defined above.

See also:

expint (page 502) Generalised exponential integral.

E1 (page 504) Special case of the generalised exponential integral.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

Si (page 507) Sine integral.

Ci (page 508) Cosine integral.

Shi (page 509) Hyperbolic sine integral.

Chi (page 510) Hyperbolic cosine integral.

`sympy.functions.special.gamma_functions.uppergamma` (page 488) Upper incomplete gamma function.

References

[R240] (page 1861), [R241] (page 1861), [R242] (page 1861)

Examples

```
>>> from sympy import Ei, polar_lift, exp_polar, I, pi
>>> from sympy.abc import x
```

```
>>> Ei(-1)
Ei(-1)
```

This yields a real value:

```
>>> Ei(-1).n(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```
>>> Ei(polar_lift(-1)).n(chop=True)
-0.21938393439552 + 3.14159265358979*I
```

The exponential integral has a logarithmic branch point at the origin:

```
>>> Ei(x*exp_polar(2*I*pi))
Ei(x) + 2*I*pi
```

Differentiation is supported:

```
>>> Ei(x).diff(x)
exp(x)/x
```

The exponential integral is related to many other special functions. For example:

```
>>> from sympy import uppergamma, expint, Shi
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)
```

class `sympy.functions.special.error_functions.expint`
Generalized exponential integral.

This function is defined as

$$E_{\nu}(z) = z^{\nu-1}\Gamma(1-\nu, z),$$

where $\Gamma(1-\nu, z)$ is the upper incomplete gamma function (`uppergamma`).

Hence for z with positive real part we have

$$E_\nu(z) = \int_1^\infty \frac{e^{-zt}}{z^\nu} dt,$$

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for $E_\nu(z)$. If ν is a non-positive integer the exponential integral is thus an unbranched function of z , otherwise there is a branch point at the origin. Refer to the incomplete gamma function documentation for details of the branching behavior.

See also:

Ei (page 501) Another related function called exponential integral.

E1 (page 504) The classical case, returns `expint(1, z)`.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

Si (page 507) Sine integral.

Ci (page 508) Cosine integral.

Shi (page 509) Hyperbolic sine integral.

Chi (page 510) Hyperbolic cosine integral.

`sympy.functions.special.gamma_functions.uppergamma` (page 488)

References

[R243] (page 1861), [R244] (page 1861), [R245] (page 1861)

Examples

```
>>> from sympy import expint, S
>>> from sympy.abc import nu, z
```

Differentiation is supported. Differentiation with respect to z explains further the name: for integral orders, the exponential integral is an iterated integral of the exponential function.

```
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to ν has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg(((), (1, 1)), ((0, 0, -nu + 1), ()), z)
```

At non-positive integer orders, the exponential integral reduces to the exponential function:

```
>>> expint(0, z)
exp(-z)/z
>>> expint(-1, z)
exp(-z)/z + exp(-z)/z**2
```

At half-integers it reduces to error functions:

```
>>> expint(S(1)/2, z)
sqrt(pi)*erfc(sqrt(z))/sqrt(z)
```

At positive integer orders it can be rewritten in terms of exponentials and `expint(1, z)`. Use `expand_func()` to do this:

```
>>> from sympy import expand_func
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + (-z**3 + z**2 - 2*z + 6)*exp(-z)/24
```

The generalised exponential integral is essentially equivalent to the incomplete gamma function:

```
>>> from sympy import uppergamma
>>> expint(nu, z).rewrite(uppergamma)
z**(nu - 1)*uppergamma(-nu + 1, z)
```

As such it is branched at the origin:

```
>>> from sympy import exp_polar, pi, I
>>> expint(4, z*exp_polar(2*pi*I))
I*pi*z**3/3 + expint(4, z)
>>> expint(nu, z*exp_polar(2*pi*I))
z**(nu - 1)*(exp(2*I*pi*nu) - 1)*gamma(-nu + 1) + expint(nu, z)
```

`sympy.functions.special.error_functions.Ei(z)`

Classical case of the generalized exponential integral.

This is equivalent to `expint(1, z)`.

See also:

Ei (page 501) Exponential integral.

expint (page 502) Generalised exponential integral.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

Si (page 507) Sine integral.

Ci (page 508) Cosine integral.

Shi (page 509) Hyperbolic sine integral.

Chi (page 510) Hyperbolic cosine integral.

class `sympy.functions.special.error_functions.li`

The classical logarithmic integral.

For the use in SymPy, this function is defined as

$$\operatorname{li}(x) = \int_0^x \frac{1}{\log(t)} dt.$$

See also:**Li** (page 506) Offset logarithmic integral.**Ei** (page 501) Exponential integral.**expint** (page 502) Generalised exponential integral.**E1** (page 504) Special case of the generalised exponential integral.**Si** (page 507) Sine integral.**Ci** (page 508) Cosine integral.**Shi** (page 509) Hyperbolic sine integral.**Chi** (page 510) Hyperbolic cosine integral.**References**[\[R246\]](#) (page 1861), [\[R247\]](#) (page 1862), [\[R248\]](#) (page 1862), [\[R249\]](#) (page 1862)**Examples**

```
>>> from sympy import I, oo, li
>>> from sympy.abc import z
```

Several special values are known:

```
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(li(z), z)
1/log(z)
```

Defining the li function via an integral:

The logarithmic integral can also be defined in terms of Ei :

```
>>> from sympy import Ei
>>> li(z).rewrite(Ei)
Ei(log(z))
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> li(2).evalf(30)
1.04516378011749278484458888919
```

```
>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```
>>> from mpmath import findroot
>>> findroot(li, 2)
1.45136923488338
```

Further transformations include rewriting li in terms of the trigonometric integrals Si , Ci , Shi and Chi :

```
>>> from sympy import Si, Ci, Shi, Chi
>>> li(z).rewrite(Si)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Ci)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Shi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
>>> li(z).rewrite(Chi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
```

class sympy.functions.special.error_functions.**Li**

The offset logarithmic integral.

For the use in SymPy, this function is defined as

$$\text{Li}(x) = \text{li}(x) - \text{li}(2)$$

See also:

li (page 504) Logarithmic integral.

Ei (page 501) Exponential integral.

expint (page 502) Generalised exponential integral.

E1 (page 504) Special case of the generalised exponential integral.

Si (page 507) Sine integral.

Ci (page 508) Cosine integral.

Shi (page 509) Hyperbolic sine integral.

Chi (page 510) Hyperbolic cosine integral.

References

[R250] (page 1862), [R251] (page 1862), [R252] (page 1862)

Examples

```
>>> from sympy import I, oo, Li
>>> from sympy.abc import z
```

The following special value is known:


```
>>> Li(2)
0
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(Li(z), z)
1/log(z)
```

The shifted logarithmic integral can be written in terms of $li(z)$:

```
>>> from sympy import li
>>> Li(z).rewrite(li)
li(z) - li(2)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> Li(2).evalf(30)
0
```

```
>>> Li(4).evalf(30)
1.92242131492155809316615998938
```

class sympy.functions.special.error_functions.**Si**
Sine integral.

This function is defined by

$$\text{Si}(z) = \int_0^z \frac{\sin t}{t} dt.$$

It is an entire function.

See also:

Ci (page 508) Cosine integral.

Shi (page 509) Hyperbolic sine integral.

Chi (page 510) Hyperbolic cosine integral.

Ei (page 501) Exponential integral.

expint (page 502) Generalised exponential integral.

sinc unnormalized sinc function

E1 (page 504) Special case of the generalised exponential integral.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

References

[R253] (page 1862)

Examples

```
>>> from sympy import Si
>>> from sympy.abc import z
```

The sine integral is an antiderivative of $\sin(z)/z$:

```
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Si(z*exp_polar(2*I*pi))
Si(z)
```

Sine integral behaves much like ordinary sine under multiplication by I :

```
>>> Si(I*z)
I*Shi(z)
>>> Si(-z)
-Si(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Si(z).rewrite(expint)
-I*(-expint(1, z*exp_polar(-I*pi/2))/2 +
    expint(1, z*exp_polar(I*pi/2))/2) + pi/2
```

It can be rewritten in the form of sinc function (By definition)

```
>>> from sympy import sinc
>>> Si(z).rewrite(sinc)
Integral(sinc(t), (t, 0, z))
```

class sympy.functions.special.error_functions.Ci
Cosine integral.

This function is defined for positive x by

$$\text{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt = - \int_x^\infty \frac{\cos t}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Ci}(z) = -\frac{E_1(e^{i\pi/2}z) + E_1(e^{-i\pi/2}z)}{2}$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for $z \in \mathbb{C}$ with $\Re(z) > 0$. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

See also:

Si (page 507) Sine integral.

Shi (page 509) Hyperbolic sine integral.

Chi (page 510) Hyperbolic cosine integral.

Ei (page 501) Exponential integral.

expint (page 502) Generalised exponential integral.

E1 (page 504) Special case of the generalised exponential integral.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

References

[R254] (page 1862)

Examples

```
>>> from sympy import Ci
>>> from sympy.abc import z
```

The cosine integral is a primitive of $\cos(z)/z$:

```
>>> Ci(z).diff(z)
cos(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary $\cos$ under multiplication by i :

```
>>> from sympy import polar_lift
>>> Ci(polar_lift(I)*z)
Chi(z) + I*pi/2
>>> Ci(polar_lift(-1)*z)
Ci(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```

class `sympy.functions.special.error_functions.Shi`
Sinh integral.

This function is defined by

$$\text{Shi}(z) = \int_0^z \frac{\sinh t}{t} dt.$$

It is an entire function.

See also:

Si (page 507) Sine integral.

Ci (page 508) Cosine integral.

Chi (page 510) Hyperbolic cosine integral.

Ei (page 501) Exponential integral.

expint (page 502) Generalised exponential integral.

E1 (page 504) Special case of the generalised exponential integral.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

References

[R255] (page 1862)

Examples

```
>>> from sympy import Shi
>>> from sympy.abc import z
```

The Sinh integral is a primitive of $\sinh(z)/z$:

```
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Shi(z*exp_polar(2*I*pi))
Shi(z)
```

The sinh integral behaves much like ordinary sinh under multiplication by i :

```
>>> Shi(I*z)
I*Si(z)
>>> Shi(-z)
-Shi(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

class sympy.functions.special.error_functions.Chi
Cosh integral.

This function is defined for positive x by

$$\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Chi}(z) = \text{Ci}\left(e^{i\pi/2}z\right) - i\frac{\pi}{2},$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

See also:

Si (page 507) Sine integral.

Ci (page 508) Cosine integral.

Shi (page 509) Hyperbolic sine integral.

Ei (page 501) Exponential integral.

expint (page 502) Generalised exponential integral.

E1 (page 504) Special case of the generalised exponential integral.

li (page 504) Logarithmic integral.

Li (page 506) Offset logarithmic integral.

References

[R256] (page 1862)

Examples

```
>>> from sympy import Chi
>>> from sympy.abc import z
```

The cosh integral is a primitive of $\cosh(z)/z$:

```
>>> Chi(z).diff(z)
cosh(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```

The cosh integral behaves somewhat like ordinary cosh under multiplication by i :

```
>>> from sympy import polar_lift
>>> Chi(polar_lift(I)*z)
Ci(z) + I*pi/2
>>> Chi(polar_lift(-1)*z)
Chi(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

Bessel Type Functions

class `sympy.functions.special.bessel.BesselBase`

Abstract base class for bessel-type functions.

This class is meant to reduce code duplication. All Bessel type functions can 1) be differentiated, and the derivatives expressed in terms of similar functions and 2) be rewritten in terms of other bessel-type functions.

Here “bessel-type functions” are assumed to have one complex parameter.

To use this base class, define class attributes `_a` and `_b` such that $2 * F_n' = -_a * F_{n+1} + _b * F_{n-1}$.

argument

The argument of the bessel-type function.

order

The order of the bessel-type function.

class `sympy.functions.special.bessel.besselj`

Bessel function of the first kind.

The Bessel J function of order ν is defined to be the function satisfying Bessel’s differential equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2)w = 0,$$

with Laurent expansion

$$J_\nu(z) = z^\nu \left(\frac{1}{\Gamma(\nu + 1)2^\nu} + O(z^2) \right),$$

if ν is not a negative integer. If $\nu = -n \in \mathbb{Z}_{<0}$ is a negative integer, then the definition is

$$J_{-n}(z) = (-1)^n J_n(z).$$

See also:

[bessely](#) (page 513), [besseli](#) (page 513), [besselk](#) (page 514)

References

[R257] (page 1862), [R258] (page 1862), [R259] (page 1862), [R260] (page 1862)

Examples

Create a Bessel function object:

```
>>> from sympy import besselj, jn
>>> from sympy.abc import z, n
>>> b = besselj(n, z)
```

Differentiate it:

```
>>> b.diff(z)
besselj(n - 1, z)/2 - besselj(n + 1, z)/2
```

Rewrite in terms of spherical Bessel functions:

```
>>> b.rewrite(jn)
sqrt(2)*sqrt(z)*jn(n - 1/2, z)/sqrt(pi)
```

Access the parameter and argument:

```
>>> b.order
n
>>> b.argument
z
```

class `sympy.functions.special.bessel.bessely`

Bessel function of the second kind.

The Bessel Y function of order ν is defined as

$$Y_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{J_\mu(z) \cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

where $J_\mu(z)$ is the Bessel function of the first kind.

It is a solution to Bessel's equation, and linearly independent from J_ν .

See also:

[besselj](#) (page 512), [besseli](#) (page 513), [besselk](#) (page 514)

References

[R261] (page 1862)

Examples

```
>>> from sympy import bessely, yn
>>> from sympy.abc import z, n
>>> b = bessely(n, z)
>>> b.diff(z)
bessely(n - 1, z)/2 - bessely(n + 1, z)/2
>>> b.rewrite(yn)
sqrt(2)*sqrt(z)*yn(n - 1/2, z)/sqrt(pi)
```

class `sympy.functions.special.bessel.besseli`

Modified Bessel function of the first kind.

The Bessel I function is a solution to the modified Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 + \nu^2) w = 0.$$

It can be defined as

$$I_\nu(z) = i^{-\nu} J_\nu(iz),$$

where $J_\nu(z)$ is the Bessel function of the first kind.

See also:

[besselj](#) (page 512), [bessely](#) (page 513), [besselk](#) (page 514)

References

[R262] (page 1862)

Examples

```
>>> from sympy import besselj
>>> from sympy.abc import z, n
>>> besselj(n, z).diff(z)
besselj(n - 1, z)/2 + besselj(n + 1, z)/2
```

class sympy.functions.special.bessel.besselk

Modified Bessel function of the second kind.

The Bessel K function of order ν is defined as

$$K_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_\mu(z)}{\sin(\pi\mu)},$$

where $I_\mu(z)$ is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from Y_ν .

See also:

[besselj](#) (page 512), [besseli](#) (page 513), [bessely](#) (page 513)

References

[R263] (page 1862)

Examples

```
>>> from sympy import besselk
>>> from sympy.abc import z, n
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

class sympy.functions.special.bessel.hankel1

Hankel function of the first kind.

This function is defined as

$$H_\nu^{(1)} = J_\nu(z) + iY_\nu(z),$$

where $J_\nu(z)$ is the Bessel function of the first kind, and $Y_\nu(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation.

See also:

[hankel2](#) (page 515), [besselj](#) (page 512), [bessely](#) (page 513)

References

[R264] (page 1862)

Examples

```
>>> from sympy import hankel1
>>> from sympy.abc import z, n
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

class sympy.functions.special.bessel.hankel2

Hankel function of the second kind.

This function is defined as

$$H_{\nu}^{(2)} = J_{\nu}(z) - iY_{\nu}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind, and $Y_{\nu}(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation, and linearly independent from $H_{\nu}^{(1)}$.

See also:

[hankel1](#) (page 514), [besselj](#) (page 512), [bessely](#) (page 513)

References

[R265] (page 1862)

Examples

```
>>> from sympy import hankel2
>>> from sympy.abc import z, n
>>> hankel2(n, z).diff(z)
hankel2(n - 1, z)/2 - hankel2(n + 1, z)/2
```

class sympy.functions.special.bessel.jn

Spherical Bessel function of the first kind.

This function is a solution to the spherical Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + 2z \frac{dw}{dz} + (z^2 - \nu(\nu + 1))w = 0.$$

It can be defined as

$$j_{\nu}(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+\frac{1}{2}}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind.

The spherical Bessel functions of integral order are calculated using the formula:

$$j_n(z) = f_n(z) \sin z + (-1)^{n+1} f_{-n-1}(z) \cos z,$$

where the coefficients $f_n(z)$ are available as `polys.orthopolys.spherical_bessel_fn()`.

See also:

[besselj](#) (page 512), [bessely](#) (page 513), [besselk](#) (page 514), [yn](#) (page 516)

References

[R266] (page 1862)

Examples

```
>>> from sympy import Symbol, jn, sin, cos, expand_func, besselj, bessely
>>> from sympy import simplify
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(jn(0, z)))
sin(z)/z
>>> expand_func(jn(1, z)) == sin(z)/z**2 - cos(z)/z
True
>>> expand_func(jn(3, z))
(-6/z**2 + 15/z**4)*sin(z) + (1/z - 15/z**3)*cos(z)
>>> jn(nu, z).rewrite(besselj)
sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(nu + 1/2, z)/2
>>> jn(nu, z).rewrite(bessely)
(-1)**nu*sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(-nu - 1/2, z)/2
>>> jn(2, 5.2+0.3j).evalf(20)
0.099419756723640344491 - 0.054525080242173562897*I
```

class `sympy.functions.special.bessel.yn`

Spherical Bessel function of the second kind.

This function is another solution to the spherical Bessel equation, and linearly independent from j_n . It can be defined as

$$y_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+\frac{1}{2}}(z),$$

where $Y_\nu(z)$ is the Bessel function of the second kind.

For integral orders n , y_n is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

See also:

[besselj](#) (page 512), [bessely](#) (page 513), [besselk](#) (page 514), [jn](#) (page 515)

References

[R267] (page 1862)

Examples

```
>>> from sympy import Symbol, yn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(yn(0, z)))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2-sin(z)/z
True
>>> yn(nu, z).rewrite(besselj)
(-1)**(nu + 1)*sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(-nu - 1/2, z)/2
>>> yn(nu, z).rewrite(bessely)
sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(nu + 1/2, z)/2
>>> yn(2, 5.2+0.3j).evalf(20)
0.18525034196069722536 + 0.014895573969924817587*I
```

`sympy.functions.special.bessel.jn_zeros(n, k, method='sympy', dps=15)`
 Zeros of the spherical Bessel function of the first kind.

This returns an array of zeros of `jn` up to the `k`-th zero.

- `method = "sympy"`: uses `mpmath.besseljzero()`
- `method = "scipy"`: uses the SciPy's `sph_jn` and `newton` to find all roots, which is faster than computing the zeros using a general numerical solver, but it requires SciPy and only works with low precision floating point numbers. [The function used with `method="sympy"` is a recent addition to `mpmath`, before that a general solver was used.]

See also:

`jn` (page 515), `yn` (page 516), `besselj` (page 512), `besselk` (page 514), `bessely` (page 513)

Examples

```
>>> from sympy import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

Airy Functions

class `sympy.functions.special.bessel.AiryBase`

Abstract base class for Airy functions.

This class is meant to reduce code duplication.

class `sympy.functions.special.bessel.airyai`

The Airy function Ai of the first kind.

The Airy function $Ai(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{d^2 w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$Ai(z) := \frac{1}{\pi} \int_0^\infty \cos\left(\frac{t^3}{3} + zt\right) dt.$$

See also:

airybi (page 519) Airy function of the second kind.

airyaiprime (page 520) Derivative of the Airy function of the first kind.

airybiprime (page 521) Derivative of the Airy function of the second kind.

References

[R268] (page 1862), [R269] (page 1862), [R270] (page 1862), [R271] (page 1862)

Examples

Create an Airy function object:

```
>>> from sympy import airyai
>>> from sympy.abc import z
```

```
>>> airyai(z)
airyai(z)
```

Several special values are known:

```
>>> airyai(0)
3**(1/3)/(3*gamma(2/3))
>>> from sympy import oo
>>> airyai(oo)
0
>>> airyai(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), z, 2)
z*airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyai(z), z, 0, 3)
3**(5/6)*gamma(1/3)/(6*pi) - 3**(1/6)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyai(-2).evalf(50)
0.22740742820168557599192443603787379946077222541710
```

Rewrite Ai(z) in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyai(z).rewrite(hyper)
-3**(2/3)*z*hyper(( ), (4/3, ), z**3/9)/(3*gamma(1/3)) + 3**(1/3)*hyper(( ), (2/3, ),
↪ z**3/9)/(3*gamma(2/3))
```

class sympy.functions.special.bessel.airybi

The Airy function Bi of the second kind.

The Airy function Bi(z) is defined to be the function satisfying Airy's differential equation

$$\frac{d^2 w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$\text{Bi}(z) := \frac{1}{\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} + zt\right) + \sin\left(\frac{t^3}{3} + zt\right) dt.$$

See also:

airyai (page 517) Airy function of the first kind.

airyaiprime (page 520) Derivative of the Airy function of the first kind.

airybiprime (page 521) Derivative of the Airy function of the second kind.

References

[R272] (page 1862), [R273] (page 1862), [R274] (page 1862), [R275] (page 1862)

Examples

Create an Airy function object:

```
>>> from sympy import airybi
>>> from sympy.abc import z
```

```
>>> airybi(z)
airybi(z)
```

Several special values are known:

```
>>> airybi(0)
3**(5/6)/(3*gamma(2/3))
>>> from sympy import oo
>>> airybi(oo)
oo
>>> airybi(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), z, 2)
z*airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybi(z), z, 0, 3)
3**(1/3)*gamma(1/3)/(2*pi) + 3**(2/3)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite $\text{Bi}(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybi(z).rewrite(hyper)
3**(1/6)*z*hyper(( ), (4/3, ), z**3/9)/gamma(1/3) + 3**(5/6)*hyper(( ), (2/3, ), z**3/
↪ 9)/(3*gamma(2/3))
```

class `sympy.functions.special.bessel.airyaiprime`

The derivative Ai' of the Airy function of the first kind.

The Airy function $\text{Ai}'(z)$ is defined to be the function

$$\text{Ai}'(z) := \frac{d \text{Ai}(z)}{dz}.$$

See also:

`airyai` (page 517) Airy function of the first kind.

`airybi` (page 519) Airy function of the second kind.

`airybiprime` (page 521) Derivative of the Airy function of the second kind.

References

[R276] (page 1862), [R277] (page 1862), [R278] (page 1862), [R279] (page 1862)

Examples

Create an Airy function object:

```
>>> from sympy import airyprime
>>> from sympy.abc import z
```

```
>>> airyprime(z)
airyprime(z)
```

Several special values are known:

```
>>> airyprime(0)
-3**(2/3)/(3*gamma(1/3))
>>> from sympy import oo
>>> airyprime(oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyprime(z))
airyprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyprime(z), z)
z*airyai(z)
>>> diff(airyprime(z), z, 2)
z*airyprime(z) + airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyprime(z), z, 0, 3)
-3**(2/3)/(3*gamma(1/3)) + 3**(1/3)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyprime(-2).evalf(50)
0.61825902074169104140626429133247528291577794512415
```

Rewrite $\text{Ai}'(z)$ in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyprime(z).rewrite(hyper)
3**(1/3)*z**2*hyper((), (5/3,), z**3/9)/(6*gamma(2/3)) - 3**(2/3)*hyper((), (1/3,
↪), z**3/9)/(3*gamma(1/3))
```

class sympy.functions.special.bessel.airybiprime

The derivative Bi' of the Airy function of the first kind.

The Airy function $\text{Bi}'(z)$ is defined to be the function

$$\text{Bi}'(z) := \frac{d \text{Bi}(z)}{dz}.$$

See also:

[airyai](#) (page 517) Airy function of the first kind.

airybi (page 519) Airy function of the second kind.

airyaiprime (page 520) Derivative of the Airy function of the first kind.

References

[R280] (page 1862), [R281] (page 1862), [R282] (page 1862), [R283] (page 1862)

Examples

Create an Airy function object:

```
>>> from sympy import airybiprime
>>> from sympy.abc import z
```

```
>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```
>>> airybiprime(0)
3**(1/6)/gamma(1/3)
>>> from sympy import oo
>>> airybiprime(oo)
oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airybiprime(z), z)
z*airybi(z)
>>> diff(airybiprime(z), z, 2)
z*airybiprime(z) + airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybiprime(z), z, 0, 3)
3**(1/6)/gamma(1/3) + 3**(5/6)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybiprime(-2).evalf(50)
0.27879516692116952268509756941098324140300059345163
```

Rewrite $\text{Bi}'(z)$ in terms of hypergeometric functions:


```
>>> from sympy import hyper
>>> airybiprime(z).rewrite(hyper)
3**(5/6)*z**2*hyper(( ), (5/3, ), z**3/9)/(6*gamma(2/3)) + 3**(1/6)*hyper(( ), (1/3,
↪ ), z**3/9)/gamma(1/3)
```

B-Splines

`sympy.functions.special.bsplines.bspline_basis(d, knots, n, x)`

The n -th B-spline at x of degree d with knots.

B-Splines are piecewise polynomials of degree d [R284] (page 1863). They are defined on a set of knots, which is a sequence of integers or floats.

The 0th degree splines have a value of one on a single interval:

```
>>> from sympy import bspline_basis
>>> from sympy.abc import x
>>> d = 0
>>> knots = range(5)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, (x >= 0) & (x <= 1)), (0, True))
```

For a given (d, knots) there are $\text{len}(\text{knots}) - d - 1$ B-splines defined, that are indexed by n (starting at 0).

Here is an example of a cubic B-spline:

```
>>> bspline_basis(3, range(5), 0, x)
Piecewise((x**3/6, (x >= 0) & (x <= 1)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3,
           (x >= 1) & (x <= 2)),
          (x**3/2 - 4*x**2 + 10*x - 22/3,
           (x >= 2) & (x <= 3)),
          (-x**3/6 + 2*x**2 - 8*x + 32/3,
           (x >= 3) & (x <= 4)),
          (0, True))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```
>>> d = 1
>>> knots = [0, 0, 2, 3, 4]
>>> bspline_basis(d, knots, 0, x)
Piecewise((-x/2 + 1, (x >= 0) & (x <= 2)), (0, True))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-splines many times, it is best to lambdify them first:

```
>>> from sympy import lambdify
>>> d = 3
>>> knots = range(10)
>>> b0 = bspline_basis(d, knots, 0, x)
>>> f = lambdify(x, b0)
>>> y = f(0.5)
```

See also:

`bsplines_basis_set`

References

[R284] (page 1863)

`sympy.functions.special.bsplines.bspline_basis_set(d, knots, x)`

Return the $\text{len}(\text{knots}) - d - 1$ B-splines at x of degree d with knots.

This function returns a list of Piecewise polynomials that are the $\text{len}(\text{knots}) - d - 1$ B-splines of degree d for the given knots. This function calls `bspline_basis(d, knots, n, x)` for different values of n .

See also:

`bsplines_basis`

Examples

```
>>> from sympy import bspline_basis_set
>>> from sympy.abc import x
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines
[Piecewise((x**2/2, (x >= 0) & (x <= 1)),
          (-x**2 + 3*x - 3/2, (x >= 1) & (x <= 2)),
          (x**2/2 - 3*x + 9/2, (x >= 2) & (x <= 3)),
          (0, True)),
 Piecewise((x**2/2 - x + 1/2, (x >= 1) & (x <= 2)),
          (-x**2 + 5*x - 11/2, (x >= 2) & (x <= 3)),
          (x**2/2 - 4*x + 8, (x >= 3) & (x <= 4)),
          (0, True))]
```

Riemann Zeta and Related Functions

class `sympy.functions.special.zeta_functions.zeta`

Hurwitz zeta function (or Riemann zeta function).

For $\text{Re}(a) > 0$ and $\text{Re}(s) > 1$, this function is defined as

$$\zeta(s, a) = \sum_{n=0}^{\infty} \frac{1}{(n+a)^s},$$

where the standard choice of argument for $n + a$ is used. For fixed a with $\text{Re}(a) > 0$ the Hurwitz zeta function admits a meromorphic continuation to all of $\mathbb{C}$, it is an unbranched function with a simple pole at $s = 1$.

Analytic continuation to other a is possible under some circumstances, but this is not typically done.

The Hurwitz zeta function is a special case of the Lerch transcendent:

$$\zeta(s, a) = \Phi(1, s, a).$$

This formula defines an analytic continuation for all possible values of s and a (also $\text{Re}(a) < 0$), see the documentation of `lerchphi` (page 527) for a description of the branching behavior.

If no value is passed for a , by this function assumes a default value of $a = 1$, yielding the Riemann zeta function.

See also:

[dirichlet_eta](#) (page 526), [lerchphi](#) (page 527), [polylog](#) (page 526)

References

[R285] (page 1863), [R286] (page 1863)

Examples

For $a = 1$ the Hurwitz zeta function reduces to the famous Riemann zeta function:

$$\zeta(s, 1) = \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

```
>>> from sympy import zeta
>>> from sympy.abc import s
>>> zeta(s, 1)
zeta(s)
>>> zeta(s)
zeta(s)
```

The Riemann zeta function can also be expressed using the Dirichlet eta function:

```
>>> from sympy import dirichlet_eta
>>> zeta(s).rewrite(dirichlet_eta)
dirichlet_eta(s)/(-2**(-s + 1) + 1)
```

The Riemann zeta function at positive even integer and negative odd integer values is related to the Bernoulli numbers:

```
>>> zeta(2)
pi**2/6
>>> zeta(4)
pi**4/90
>>> zeta(-1)
-1/12
```

The specific formulae are:

$$\zeta(2n) = (-1)^{n+1} \frac{B_{2n}(2\pi)^{2n}}{2(2n)!}$$

$$\zeta(-n) = -\frac{B_{n+1}}{n+1}$$

At negative even integers the Riemann zeta function is zero:

```
>>> zeta(-4)
0
```

No closed-form expressions are known at positive odd integers, but numerical evaluation is possible:

```
>>> zeta(3).n()
1.20205690315959
```

The derivative of $\zeta(s, a)$ with respect to a is easily computed:

```
>>> from sympy.abc import a
>>> zeta(s, a).diff(a)
-s*zeta(s + 1, a)
```

However the derivative with respect to s has no useful closed form expression:

```
>>> zeta(s, a).diff(s)
Derivative(zeta(s, a), s)
```

The Hurwitz zeta function can be expressed in terms of the Lerch transcendent, `sympy.functions.special.lerchphi`:

```
>>> from sympy import lerchphi
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

class `sympy.functions.special.zeta_functions.dirichlet_eta`
Dirichlet eta function.

For $\operatorname{Re}(s) > 0$, this function is defined as

$$\eta(s) = \sum_{n=1}^{\infty} \frac{(-1)^n}{n^s}.$$

It admits a unique analytic continuation to all of $\mathbb{C}$. It is an entire, unbranched function.

See also:

[zeta](#) (page 524)

References

[R287] (page 1863)

Examples

The Dirichlet eta function is closely related to the Riemann zeta function:

```
>>> from sympy import dirichlet_eta, zeta
>>> from sympy.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(-2*(-s + 1) + 1)*zeta(s)
```

class `sympy.functions.special.zeta_functions.polylog`
Polylogarithm function.

For $|z| < 1$ and $s \in \mathbb{C}$, the polylogarithm is defined by

$$\operatorname{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for n . It admits an analytic continuation which is branched at $z = 1$ (notably not on the sheet of initial definition), $z = 0$ and $z = \infty$.

The name polylogarithm comes from the fact that for $s = 1$, the polylogarithm is related to the ordinary logarithm (see examples), and that

$$\text{Li}_{s+1}(z) = \int_0^z \frac{\text{Li}_s(t)}{t} dt.$$

The polylogarithm is a special case of the Lerch transcendent:

$$\text{Li}_s(z) = z\Phi(z, s, 1)$$

See also:

[zeta](#) (page 524), [lerchphi](#) (page 527)

Examples

For $z \in \{0, 1, -1\}$, the polylogarithm is automatically expressed using other functions:

```
>>> from sympy import polylog
>>> from sympy.abc import s
>>> polylog(s, 0)
0
>>> polylog(s, 1)
zeta(s)
>>> polylog(s, -1)
-dirichlet_eta(s)
```

If s is a negative integer, 0 or 1, the polylogarithm can be expressed using elementary functions. This can be done using `expand_func()`:

```
>>> from sympy import expand_func
>>> from sympy.abc import z
>>> expand_func(polylog(1, z))
-log(-z + 1)
>>> expand_func(polylog(0, z))
z/(-z + 1)
```

The derivative with respect to z can be computed in closed form:

```
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```
>>> from sympy import lerchphi
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

class `sympy.functions.special.zeta_functions.lerchphi`
Lerch transcendent (Lerch phi function).

For $\text{Re}(a) > 0$, $|z| < 1$ and $s \in \mathbb{C}$, the Lerch transcendent is defined as

$$\Phi(z, s, a) = \sum_{n=0}^{\infty} \frac{z^n}{(n+a)^s},$$

where the standard branch of the argument is used for $n+a$, and by analytic continuation for other values of the parameters.

A commonly used related function is the Lerch zeta function, defined by

$$L(q, s, a) = \Phi(e^{2\pi i q}, s, a).$$

Analytic Continuation and Branching Behavior

It can be shown that

$$\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}.$$

This provides the analytic continuation to $\operatorname{Re}(a) \leq 0$.

Assume now $\operatorname{Re}(a) > 0$. The integral representation

$$\Phi_0(z, s, a) = \int_0^\infty \frac{t^{s-1} e^{-at}}{1 - ze^{-t}} \frac{dt}{\Gamma(s)}$$

provides an analytic continuation to $\mathbb{C} - [1, \infty)$. Finally, for $x \in (1, \infty)$ we find

$$\lim_{\epsilon \rightarrow 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \rightarrow 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},$$

using the standard branch for both $\log x$ and $\log \log x$ (a branch of $\log \log x$ is needed to evaluate $\log x^{s-1}$). This concludes the analytic continuation. The Lerch transcendent is thus branched at $z \in \{0, 1, \infty\}$ and $a \in \mathbb{Z}_{\leq 0}$. For fixed z, a outside these branch points, it is an entire function of s .

See also:

[polylog](#) (page 526), [zeta](#) (page 524)

References

[R288] (page 1863), [R289] (page 1863), [R290] (page 1863)

Examples

The Lerch transcendent is a fairly general function, for this reason it does not automatically evaluate to simpler functions. Use `expand_func()` to achieve this.

If $z = 1$, the Lerch transcendent reduces to the Hurwitz zeta function:

```
>>> from sympy import lerchphi, expand_func
>>> from sympy.abc import z, s, a
>>> expand_func(lerchphi(1, s, a))
zeta(s, a)
```

More generally, if z is a root of unity, the Lerch transcendent reduces to a sum of Hurwitz zeta functions:

```
>>> expand_func(lerchphi(-1, s, a))
2**(-s)*zeta(s, a/2) - 2**(-s)*zeta(s, a/2 + 1/2)
```

If $a = 1$, the Lerch transcendent reduces to the polylogarithm:

```
>>> expand_func(lerchphi(z, s, 1))
polylog(s, z)/z
```

More generally, if a is rational, the Lerch transcendent reduces to a sum of polylogarithms:

```
>>> from sympy import S
>>> expand_func(lerchphi(z, s, S(1)/2))
2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
            polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))
>>> expand_func(lerchphi(z, s, S(3)/2))
-2*s/z + 2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
                    polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))/z
```

The derivatives with respect to z and a can be computed in closed form:

```
>>> lerchphi(z, s, a).diff(z)
(-a*lerchphi(z, s, a) + lerchphi(z, s - 1, a))/z
>>> lerchphi(z, s, a).diff(a)
-s*lerchphi(z, s + 1, a)
```

class sympy.functions.special.zeta_functions.stieltjes

Represents Stieltjes constants, γ_k that occur in Laurent Series expansion of the Riemann zeta function.

References

[R291] (page 1863)

Examples

```
>>> from sympy import stieltjes
>>> from sympy.abc import n, m
>>> stieltjes(n)
stieltjes(n)
```

zero'th stieltjes constant

```
>>> stieltjes(0)
EulerGamma
>>> stieltjes(0, 1)
EulerGamma
```

For generalized stieltjes constants

```
>>> stieltjes(n, m)
stieltjes(n, m)
```

Constants are only defined for integers ≥ 0

```
>>> stieltjes(-1)
zoo
```

Hypergeometric Functions

`class sympy.functions.special.hyper.hyper`

The (generalized) hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index. When convergent, it is continued analytically to the largest possible domain.

The hypergeometric function depends on two vectors of parameters, called the numerator parameters a_p , and the denominator parameters b_q . It also has an argument z . The series definition is

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!},$$

where $(a)_n = (a)(a+1) \cdots (a+n-1)$ denotes the rising factorial.

If one of the b_q is a non-positive integer then the series is undefined unless one of the a_p is a larger (i.e. smaller in magnitude) non-positive integer. If none of the b_q is a non-positive integer and one of the a_p is a non-positive integer, then the series reduces to a polynomial. To simplify the following discussion, we assume that none of the a_p or b_q is a non-positive integer. For more details, see the references.

The series converges for all z if $p \leq q$, and thus defines an entire single-valued function in this case. If $p = q + 1$ the series converges for $|z| < 1$, and can be continued analytically into a half-plane. If $p > q + 1$ the series is divergent for all z .

Note: The hypergeometric function constructor currently does not check if the parameters actually yield a well-defined function.

See also:

`sympy.simplify.hyperexpand` (page 1176), `sympy.functions.special.gamma_functions.gamma` (page 483), `meijerg` (page 532)

References

[R292] (page 1863), [R293] (page 1863)

Examples

The parameters a_p and b_q can be passed as arbitrary iterables, for example:

```
>>> from sympy.functions import hyper
>>> from sympy.abc import x, n, a
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing (it looks better using unicode):

```
>>> from sympy import pprint
>>> pprint(hyper((1, 2, 3), [3, 4], x), use_unicode=False)

  -
  |  /1, 2, 3 | \
  |  |      | x|
3  2 \ 3, 4 | /
```

The parameters must always be iterables, even if they are vectors of length one or zero:


```
>>> hyper((1, ), [], x)
hyper((1,), (), x)
```

But of course they may be variables (but if they depend on x then you should not expect much implemented functionality):

```
>>> hyper((n, a), (n**2,), x)
hyper((n, a), (n**2,), x)
```

The hypergeometric function generalizes many named special functions. The function `hyperexpand()` tries to express a hypergeometric function using named special functions. For example:

```
>>> from sympy import hyperexpand
>>> hyperexpand(hyper([], [], x))
exp(x)
```

You can also use `expand_func`:

```
>>> from sympy import expand_func
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```
>>> from sympy import S
>>> hyperexpand(hyper([], [S(1)/2], -x**2/4))
cos(x)
>>> hyperexpand(x*hyper([S(1)/2, S(1)/2], [S(3)/2], x**2))
asin(x)
```

We can also sometimes hyperexpand parametric functions:

```
>>> from sympy.abc import a
>>> hyperexpand(hyper([-a], [], x))
(-x + 1)**a
```

ap

Numerator parameters of the hypergeometric function.

argument

Argument of the hypergeometric function.

bq

Denominator parameters of the hypergeometric function.

convergence_statement

Return a condition on z under which the series converges.

eta

A quantity related to the convergence of the series.

radius_of_convergence

Compute the radius of convergence of the defining series.

Note that even if this is not ∞ , the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

```

>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyper((1, 2), [3], z).radius_of_convergence
1
>>> hyper((1, 2, 3), [4], z).radius_of_convergence
0
>>> hyper((1, 2), (3, 4), z).radius_of_convergence
oo

```

class sympy.functions.special.hyper.meijerg

The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.

The Meijer G-function depends on four sets of parameters. There are “numerator parameters” $a_1, \dots, a_n$ and $a_{n+1}, \dots, a_p$, and there are “denominator parameters” $b_1, \dots, b_m$ and $b_{m+1}, \dots, b_q$. Confusingly, it is traditionally denoted as follows (note the position of m, n, p, q , and how they relate to the lengths of the four parameter vectors):

$$G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_n & a_{n+1}, \dots, a_p \\ b_1, \dots, b_m & b_{m+1}, \dots, b_q \end{matrix} \middle| z \right).$$

However, in sympy the four parameter vectors are always available separately (see examples), so that there is no need to keep track of the decorating sub- and super-scripts on the G symbol.

The G function is defined as the following integral:

$$\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where $\Gamma(z)$ is the gamma function. There are three possible contours which we will not describe in detail here (see the references). If the integral converges along more than one of them the definitions agree. The contours all separate the poles of $\Gamma(1 - a_j + s)$ from the poles of $\Gamma(b_k - s)$, so in particular the G function is undefined if $a_j - b_k \in \mathbb{Z}_{>0}$ for some $j \leq n$ and $k \leq m$.

The conditions under which one of the contours yields a convergent integral are complicated and we do not state them here, see the references.

Note: Currently the Meijer G-function constructor does not check any convergence conditions.

See also:

[hyper](#) (page 530), [sympy.simplify.hyperexpand](#) (page 1176)

References

[R294] (page 1863), [R295] (page 1863)

Examples

You can pass the parameters either as four separate vectors:

```
>>> from sympy.functions import meijerg
>>> from sympy.abc import x, a
>>> from sympy.core.containers import Tuple
>>> from sympy import pprint
>>> pprint(meijerg((1, 2), (a, 4), (5,), [], x), use_unicode=False)
  1, 2 / 1, 2  a, 4 | \
 /--|         | x|
 \--| 4, 1 \ 5      | /
```

or as two nested vectors:

```
>>> pprint(meijerg([(1, 2), (3, 4)], ([5], Tuple()), x), use_unicode=False)
  1, 2 / 1, 2  3, 4 | \
 /--|         | x|
 \--| 4, 1 \ 5      | /
```

As with the hypergeometric function, the parameters may be passed as arbitrary iterables. Vectors of length zero and one also have to be passed as iterables. The parameters need not be constants, but if they depend on the argument then not much implemented functionality should be expected.

All the subvectors of parameters are available:

```
>>> from sympy import pprint
>>> g = meijerg([1], [2], [3], [4], x)
>>> pprint(g, use_unicode=False)
  1, 1 / 1  2 | \
 /--|         | x|
 \--| 2, 2 \ 3  4 | /
>>> g.an
(1,)
>>> g.ap
(1, 2)
>>> g.aother
(2,)
>>> g.bm
(3,)
>>> g.bq
(3, 4)
>>> g.bothr
(4,)
```

The Meijer G-function generalizes the hypergeometric functions. In some cases it can be expressed in terms of hypergeometric functions, using Slater's theorem. For example:

```
>>> from sympy import hyperexpand
>>> from sympy.abc import a, b, c
>>> hyperexpand(meijerg([a], [], [c], [b], x), allow_hyper=True)
x**c*gamma(-a + c + 1)*hyper((-a + c + 1,),
                             (-b + c + 1,), -x)/gamma(-b + c + 1)
```

Thus the Meijer G-function also subsumes many named functions as special cases. You can use `expand_func` or `hyperexpand` to (try to) rewrite a Meijer G-function in terms of named special functions. For example:

```
>>> from sympy import expand_func, S
>>> expand_func(meijerg([], [], [[0], []], -x))
```

(continues on next page)

(continued from previous page)

```
exp(x)
>>> hyperexpand(meijerg([], [], [[S(1)/2], [0]], (x/2)**2))
sin(x)/sqrt(pi)
```

an

First set of numerator parameters.

aother

Second set of numerator parameters.

ap

Combined numerator parameters.

argument

Argument of the Meijer G-function.

bm

First set of denominator parameters.

bother

Second set of denominator parameters.

bq

Combined denominator parameters.

delta

A quantity related to the convergence region of the integral, c.f. references.

get_period()Return a number P such that $G(x \cdot \exp(I \cdot P)) = G(x)$.

```
>>> from sympy.functions.special.hyper import meijerg
>>> from sympy.abc import z
>>> from sympy import pi, S
```

```
>>> meijerg([1], [], [], [], z).get_period()
2*pi
>>> meijerg([pi], [], [], [], z).get_period()
oo
>>> meijerg([1, 2], [], [], [], z).get_period()
oo
>>> meijerg([1, 1], [2], [1, S(1)/2, S(1)/3], [1], z).get_period()
12*pi
```

integrand(s)

Get the defining integrand D(s).

is_number

Returns true if expression has numeric data only.

nu

A quantity related to the convergence region of the integral, c.f. references.

Elliptic integrals

class `sympy.functions.special.elliptic_integrals.elliptic_k`

The complete elliptic integral of the first kind, defined by

$$K(m) = F\left(\frac{\pi}{2} \middle| m\right)$$

where $F(z|m)$ is the Legendre incomplete elliptic integral of the first kind.

The function $K(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

See also:

[elliptic_f](#) (page 535)

References

[R296] (page 1863), [R297] (page 1863)

Examples

```
>>> from sympy import elliptic_k, I, pi
>>> from sympy.abc import m
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(m).series(n=3)
pi/2 + pi*m/8 + 9*pi*m**2/128 + O(m**3)
```

class `sympy.functions.special.elliptic_integrals.elliptic_f`

The Legendre incomplete elliptic integral of the first kind, defined by

$$F(z|m) = \int_0^z \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

This function reduces to a complete elliptic integral of the first kind, $K(m)$, when $z = \pi/2$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

See also:

[elliptic_k](#) (page 534)

References

[R298] (page 1863), [R299] (page 1863)

Examples

```
>>> from sympy import elliptic_f, I, 0
>>> from sympy.abc import z, m
>>> elliptic_f(z, m).series(z)
```

(continues on next page)

(continued from previous page)

```

z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + 0(z**6)
>>> elliptic_f(3.0 + I/2, 1.0 + I)
2.909449841483 + 1.74720545502474*I

```

class sympy.functions.special.elliptic_integrals.elliptic_e

Called with two arguments z and m , evaluates the incomplete elliptic integral of the second kind, defined by

$$E(z|m) = \int_0^z \sqrt{1 - m \sin^2 t} dt$$

Called with a single argument m , evaluates the Legendre complete elliptic integral of the second kind

$$E(m) = E\left(\frac{\pi}{2}|m\right)$$

The function $E(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

References

[R300] (page 1863), [R301] (page 1863), [R302] (page 1863)

Examples

```

>>> from sympy import elliptic_e, I, pi, 0
>>> from sympy.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + 0(z**6)
>>> elliptic_e(m).series(n=4)
pi/2 - pi*m/8 - 3*pi*m**2/128 - 5*pi*m**3/512 + 0(m**4)
>>> elliptic_e(1 + I, 2 - I/2).n()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I

```

class sympy.functions.special.elliptic_integrals.elliptic_pi

Called with three arguments n , z and m , evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$\Pi(n; z|m) = \int_0^z \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}}$$

Called with two arguments n and m , evaluates the complete elliptic integral of the third kind:

$$\Pi(n|m) = \Pi\left(n; \frac{\pi}{2}|m\right)$$

Note that our notation defines the incomplete elliptic integral in terms of the parameter m instead of the elliptic modulus (eccentricity) k . In this case, the parameter m is defined as $m = k^2$.

References

[R303] (page 1863), [R304] (page 1863), [R305] (page 1863)

Examples

```
>>> from sympy import elliptic_pi, I, pi, 0, S
>>> from sympy.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + O(z**4)
>>> elliptic_pi(0.5 + I, 1.0 - I, 1.2)
2.50232379629182 - 0.760939574180767*I
>>> elliptic_pi(0, 0)
pi/2
>>> elliptic_pi(1.0 - I/3, 2.0 + I)
3.29136443417283 + 0.32555634906645*I
```

Mathieu Functions

class sympy.functions.special.mathieu_functions.MathieuBase

Abstract base class for Mathieu functions.

This class is meant to reduce code duplication.

class sympy.functions.special.mathieu_functions.mathieus

The Mathieu Sine function $S(a, q, z)$. This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Cosine function.

See also:

mathieuc (page 538) Mathieu cosine function.

mathieusprime (page 538) Derivative of Mathieu sine function.

mathieucprime (page 539) Derivative of Mathieu cosine function.

References

[R306] (page 1863), [R307] (page 1863), [R308] (page 1863), [R309] (page 1863)

Examples

```
>>> from sympy import diff, mathieus
>>> from sympy.abc import a, q, z
```

```
>>> mathieus(a, q, z)
mathieus(a, q, z)
```

```
>>> mathieus(a, 0, z)
sin(sqrt(a)*z)
```

```
>>> diff(mathieus(a, q, z), z)
mathieusprime(a, q, z)
```

class sympy.functions.special.mathieu_functions.mathieuc

The Mathieu Cosine function $C(a, q, z)$. This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Sine function.

See also:

mathieus (page 537) Mathieu sine function

mathieusprime (page 538) Derivative of Mathieu sine function

mathieucprime (page 539) Derivative of Mathieu cosine function

References

[R310] (page 1863), [R311] (page 1863), [R312] (page 1863), [R313] (page 1863)

Examples

```
>>> from sympy import diff, mathieuc
>>> from sympy.abc import a, q, z
```

```
>>> mathieuc(a, q, z)
mathieuc(a, q, z)
```

```
>>> mathieuc(a, 0, z)
cos(sqrt(a)*z)
```

```
>>> diff(mathieuc(a, q, z), z)
mathieucprime(a, q, z)
```

class sympy.functions.special.mathieu_functions.mathieusprime

The derivative $S'(a, q, z)$ of the Mathieu Sine function. This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Cosine function.

See also:

mathieu (page 537) Mathieu sine function

mathieuc (page 538) Mathieu cosine function

mathieucprime (page 539) Derivative of Mathieu cosine function

References

[R314] (page 1863), [R315] (page 1863), [R316] (page 1863), [R317] (page 1863)

Examples

```
>>> from sympy import diff, mathieusprime
>>> from sympy.abc import a, q, z
```

```
>>> mathieusprime(a, q, z)
mathieusprime(a, q, z)
```

```
>>> mathieusprime(a, 0, z)
sqrt(a)*cos(sqrt(a)*z)
```

```
>>> diff(mathieusprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieus(a, q, z)
```

class sympy.functions.special.mathieu_functions.mathieucprime

The derivative $C'(a, q, z)$ of the Mathieu Cosine function. This function is one solution of the Mathieu differential equation:

$$y(x)'' + (a - 2q \cos(2x))y(x) = 0$$

The other solution is the Mathieu Sine function.

See also:

mathieu (page 537) Mathieu sine function

mathieuc (page 538) Mathieu cosine function

mathieusprime (page 538) Derivative of Mathieu sine function

References

[R318] (page 1863), [R319] (page 1863), [R320] (page 1863), [R321] (page 1864)

Examples

```
>>> from sympy import diff, mathieucprime
>>> from sympy.abc import a, q, z
```

```
>>> mathieucprime(a, q, z)
mathieucprime(a, q, z)
```

```
>>> mathieucprime(a, 0, z)
-sqrt(a)*sin(sqrt(a)*z)
```

```
>>> diff(mathieucprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieuc(a, q, z)
```

Orthogonal Polynomials

This module mainly implements special orthogonal polynomials.

See also `functions.combinatorial.numbers` which contains some combinatorial polynomials.

Jacobi Polynomials

class `sympy.functions.special.polynomials.jacobi`

Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$

`jacobi(n, alpha, beta, x)` gives the n th Jacobi polynomial in x , $P_n^{(\alpha,\beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1-x)^\alpha(1+x)^\beta$.

See also:

[gegenbauer](#) (page 542), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R322] (page 1864), [R323] (page 1864), [R324] (page 1864)

Examples

```
>>> from sympy import jacobi, S, conjugate, diff
>>> from sympy.abc import n,a,b,x
```

```
>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
>>> jacobi(2, a, b, x)
(a**2/8 - a*b/4 - a/8 + b**2/8 - b/8 + x**2*(a**2/8 + a*b/4 + 7*a/8 +
b**2/8 + 7*b/8 + 3/2) + x*(a**2/4 + 3*a/4 - b**2/4 - 3*b/4) - 1/2)
```

```
>>> jacobi(n, a, b, x)
jacobi(n, a, b, x)
```

```
>>> jacobi(n, a, a, x)
RisingFactorial(a + 1, n)*gegenbauer(n,
a + 1/2, x)/RisingFactorial(2*a + 1, n)
```

```
>>> jacobi(n, 0, 0, x)
legendre(n, x)
```

```
>>> jacobi(n, S(1)/2, S(1)/2, x)
RisingFactorial(3/2, n)*chebyshevu(n, x)/factorial(n + 1)
```

```
>>> jacobi(n, -S(1)/2, -S(1)/2, x)
RisingFactorial(1/2, n)*chebyshevt(n, x)/factorial(n)
```

```
>>> jacobi(n, a, b, -x)
(-1)**n*jacobi(n, b, a, x)
```

```
>>> jacobi(n, a, b, 0)
2**(-n)*gamma(a + n + 1)*hyper((-b - n, -n), (a + 1,), -1)/(factorial(n)*gamma(a
↪ + 1))
>>> jacobi(n, a, b, 1)
RisingFactorial(a + 1, n)/factorial(n)
```

```
>>> conjugate(jacobi(n, a, b, x))
jacobi(n, conjugate(a), conjugate(b), conjugate(x))
```

```
>>> diff(jacobi(n,a,b,x), x)
(a/2 + b/2 + n/2 + 1/2)*jacobi(n - 1, a + 1, b + 1, x)
```

`sympy.functions.special.polynomials.jacobi_normalized(n, a, b, x)`

Jacobi polynomial $P_n^{(\alpha, \beta)}(x)$

`jacobi_normalized(n, alpha, beta, x)` gives the n th Jacobi polynomial in x , $P_n^{(\alpha, \beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1 - x)^\alpha (1 + x)^\beta$.

This functions returns the polynomials normilzed:

$$\int_{-1}^1 P_m^{(\alpha, \beta)}(x) P_n^{(\alpha, \beta)}(x) (1 - x)^\alpha (1 + x)^\beta dx = \delta_{m, n}$$

See also:

[gegenbauer](#) (page 542), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R325] (page 1864), [R326] (page 1864), [R327] (page 1864)

Examples

```
>>> from sympy import jacobi_normalized
>>> from sympy.abc import n,a,b,x
```

```
>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2***(a + b + 1)*gamma(a + n + 1)*gamma(b + n + 1)/((a + b + 1)*factorial(n)*gamma(a + b + n + 1)))
```

Gegenbauer Polynomials

class sympy.functions.special.polynomials.gegenbauer

Gegenbauer polynomial $C_n^{(\alpha)}(x)$

gegenbauer(n, alpha, x) gives the nth Gegenbauer polynomial in x, $C_n^{(\alpha)}(x)$.

The Gegenbauer polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1 - x^2)^{\alpha - \frac{1}{2}}$.

See also:

[jacobi](#) (page 540), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R328] (page 1864), [R329] (page 1864), [R330] (page 1864)

Examples

```
>>> from sympy import gegenbauer, conjugate, diff
>>> from sympy.abc import n,a,x
>>> gegenbauer(0, a, x)
1
>>> gegenbauer(1, a, x)
2*a*x
>>> gegenbauer(2, a, x)
-a + x**2*(2*a**2 + 2*a)
>>> gegenbauer(3, a, x)
x**3*(4*a**3/3 + 4*a**2 + 8*a/3) + x*(-2*a**2 - 2*a)
```

```
>>> gegenbauer(n, a, x)
gegenbauer(n, a, x)
>>> gegenbauer(n, a, -x)
(-1)**n*gegenbauer(n, a, x)
```

```
>>> gegenbauer(n, a, 0)
2**n*sqrt(pi)*gamma(a + n/2)/(gamma(a)*gamma(-n/2 + 1/2)*gamma(n + 1))
>>> gegenbauer(n, a, 1)
gamma(2*a + n)/(gamma(2*a)*gamma(n + 1))
```

```
>>> conjugate(gegenbauer(n, a, x))
gegenbauer(n, conjugate(a), conjugate(x))
```

```
>>> diff(gegenbauer(n, a, x), x)
2*a*gegenbauer(n - 1, a + 1, x)
```

Chebyshev Polynomials

class `sympy.functions.special.polynomials.chebyshevt`

Chebyshev polynomial of the first kind, $T_n(x)$

`chebyshevt(n, x)` gives the n th Chebyshev polynomial (of the first kind) in x , $T_n(x)$.

The Chebyshev polynomials of the first kind are orthogonal on $[-1, 1]$ with respect to the weight $\frac{1}{\sqrt{1-x^2}}$.

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R331] (page 1864), [R332] (page 1864), [R333] (page 1864), [R334] (page 1864), [R335] (page 1864)

Examples

```
>>> from sympy import chebyshevt, chebyshevu, diff
>>> from sympy.abc import n, x
>>> chebyshevt(0, x)
1
>>> chebyshevt(1, x)
x
>>> chebyshevt(2, x)
2*x**2 - 1
```

```
>>> chebyshevt(n, x)
chebyshevt(n, x)
>>> chebyshevt(n, -x)
(-1)**n*chebyshevt(n, x)
>>> chebyshevt(-n, x)
chebyshevt(n, x)
```

```
>>> chebyshevt(n, 0)
cos(pi*n/2)
>>> chebyshevt(n, -1)
(-1)**n
```

```
>>> diff(chebyshevt(n, x), x)
n*chebyshevu(n - 1, x)
```

class `sympy.functions.special.polynomials.chebyshevu`Chebyshev polynomial of the second kind, $U_n(x)$ `chebyshevu(n, x)` gives the n th Chebyshev polynomial of the second kind in x , $U_n(x)$.The Chebyshev polynomials of the second kind are orthogonal on $[-1, 1]$ with respect to the weight $\sqrt{1 - x^2}$.**See also:**

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[\[R336\]](#) (page 1864), [\[R337\]](#) (page 1864), [\[R338\]](#) (page 1864), [\[R339\]](#) (page 1864), [\[R340\]](#) (page 1864)

Examples

```
>>> from sympy import chebyshevt, chebyshevu, diff
>>> from sympy.abc import n,x
>>> chebyshevu(0, x)
1
>>> chebyshevu(1, x)
2*x
>>> chebyshevu(2, x)
4*x**2 - 1
```

```
>>> chebyshevu(n, x)
chebyshevu(n, x)
>>> chebyshevu(n, -x)
```

(continues on next page)

(continued from previous page)

```
(-1)**n*chebyshevu(n, x)
>>> chebyshevu(-n, x)
-chebyshevu(n - 2, x)
```

```
>>> chebyshevu(n, 0)
cos(pi*n/2)
>>> chebyshevu(n, 1)
n + 1
```

```
>>> diff(chebyshevu(n, x), x)
(-x*chebyshevu(n, x) + (n + 1)*chebyshevt(n + 1, x))/(x**2 - 1)
```

class sympy.functions.special.polynomials.chebyshevt_root

chebyshevt_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if $0 \leq k < n$, `chebyshevt(n, chebyshevt_root(n, k)) == 0`.

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

Examples

```
>>> from sympy import chebyshevt, chebyshevt_root
>>> chebyshevt_root(3, 2)
-sqrt(3)/2
>>> chebyshevt(3, chebyshevt_root(3, 2))
0
```

class sympy.functions.special.polynomials.chebyshevu_root

chebyshevu_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if $0 \leq k < n$, `chebyshevu(n, chebyshevu_root(n, k)) == 0`.

See also:

[chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

Examples

```
>>> from sympy import chebyshevu, chebyshevu_root
>>> chebyshevu_root(3, 2)
-sqrt(2)/2
>>> chebyshevu(3, chebyshevu_root(3, 2))
0
```

Legendre Polynomials

class sympy.functions.special.polynomials.**Legendre**

legendre(n, x) gives the nth Legendre polynomial of x, $P_n(x)$

The Legendre polynomials are orthogonal on $[-1, 1]$ with respect to the constant weight 1. They satisfy $P_n(1) = 1$ for all n; further, P_n is odd for odd n and even for even n.

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R341] (page 1864), [R342] (page 1864), [R343] (page 1864), [R344] (page 1864)

Examples

```
>>> from sympy import legendre, diff
>>> from sympy.abc import x, n
>>> legendre(0, x)
1
>>> legendre(1, x)
x
>>> legendre(2, x)
3*x**2/2 - 1/2
>>> legendre(n, x)
legendre(n, x)
>>> diff(legendre(n,x), x)
n*(x*legendre(n, x) - legendre(n - 1, x))/(x**2 - 1)
```

class sympy.functions.special.polynomials.**assoc_legendre**

assoc_legendre(n,m, x) gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the nth order Legendre polynomial, $P_n(x)$ in the following manner:

$$P_n^m(x) = (-1)^m (1-x^2)^{\frac{m}{2}} \frac{d^m P_n(x)}{dx^m}$$

Associated Legendre polynomial are orthogonal on $[-1, 1]$ with:

- weight = 1 for the same m , and different n .
- weight = $1/(1-x^2)$ for the same n , and different m .

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R345] (page 1864), [R346] (page 1864), [R347] (page 1864), [R348] (page 1864)

Examples

```
>>> from sympy import assoc_legendre
>>> from sympy.abc import x, m, n
>>> assoc_legendre(0,0, x)
1
>>> assoc_legendre(1,0, x)
x
>>> assoc_legendre(1,1, x)
-sqrt(-x**2 + 1)
>>> assoc_legendre(n,m,x)
assoc_legendre(n, m, x)
```

Hermite Polynomials

class `sympy.functions.special.polynomials.hermite`

`hermite(n, x)` gives the n th Hermite polynomial in x , $H_n(x)$

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight $\exp(-x^2)$.

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [laguerre](#) (page 548), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R349] (page 1864), [R350] (page 1864), [R351] (page 1864)

Examples

```
>>> from sympy import hermite, diff
>>> from sympy.abc import x, n
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

Laguerre Polynomials

class sympy.functions.special.polynomials.laguerre

Returns the n th Laguerre polynomial in x , $L_n(x)$.

Parameters n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [assoc_laguerre](#) (page 549), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R352] (page 1864), [R353] (page 1864), [R354] (page 1864), [R355] (page 1864)

Examples

```
>>> from sympy import laguerre, diff
>>> from sympy.abc import x, n
>>> laguerre(0, x)
1
```

(continues on next page)

(continued from previous page)

```
>>> laguerre(1, x)
-x + 1
>>> laguerre(2, x)
x**2/2 - 2*x + 1
>>> laguerre(3, x)
-x**3/6 + 3*x**2/2 - 3*x + 1
```

```
>>> laguerre(n, x)
laguerre(n, x)
```

```
>>> diff(laguerre(n, x), x)
-assoc_laguerre(n - 1, 1, x)
```

class `sympy.functions.special.polynomials.assoc_laguerre`

Returns the n th generalized Laguerre polynomial in x , $L_n(x)$.

Parameters n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

alpha : Expr

Arbitrary expression. For $\alpha=0$ regular Laguerre polynomials will be generated.

See also:

[jacobi](#) (page 540), [gegenbauer](#) (page 542), [chebyshevt](#) (page 543), [chebyshevt_root](#) (page 545), [chebyshevu](#) (page 544), [chebyshevu_root](#) (page 545), [legendre](#) (page 546), [assoc_legendre](#) (page 546), [hermite](#) (page 547), [laguerre](#) (page 548), [sympy.polys.orthopolys.jacobi_poly](#) (page 874), [sympy.polys.orthopolys.gegenbauer_poly](#) (page 874), [sympy.polys.orthopolys.chebyshevt_poly](#) (page 873), [sympy.polys.orthopolys.chebyshevu_poly](#) (page 874), [sympy.polys.orthopolys.hermite_poly](#) (page 874), [sympy.polys.orthopolys.legendre_poly](#) (page 874), [sympy.polys.orthopolys.laguerre_poly](#) (page 875)

References

[R356] (page 1864), [R357] (page 1864), [R358] (page 1864), [R359] (page 1865)

Examples

```
>>> from sympy import laguerre, assoc_laguerre, diff
>>> from sympy.abc import x, n, a
>>> assoc_laguerre(0, a, x)
1
>>> assoc_laguerre(1, a, x)
a - x + 1
>>> assoc_laguerre(2, a, x)
a**2/2 + 3*a/2 + x**2/2 + x*(-a - 2) + 1
>>> assoc_laguerre(3, a, x)
a**3/6 + a**2 + 11*a/6 - x**3/6 + x**2*(a/2 + 3/2) +
x*(-a**2/2 - 5*a/2 - 3) + 1
```

```
>>> assoc_laguerre(n, a, 0)
binomial(a + n, a)
```

```
>>> assoc_laguerre(n, a, x)
assoc_laguerre(n, a, x)
```

```
>>> assoc_laguerre(n, 0, x)
laguerre(n, x)
```

```
>>> diff(assoc_laguerre(n, a, x), x)
-assoc_laguerre(n - 1, a + 1, x)
```

```
>>> diff(assoc_laguerre(n, a, x), a)
Sum(assoc_laguerre(_k, a, x)/(-a + n), (_k, 0, n - 1))
```

Spherical Harmonics

class `sympy.functions.special.spherical_harmonics.Ynm`
Spherical harmonics defined as

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\varphi) P_n^m(\cos(\theta))$$

`Ynm()` gives the spherical harmonic function of order n and m in θ and φ , $Y_n^m(\theta, \varphi)$. The four parameters are as follows: $n \geq 0$ an integer and m an integer such that $-n \leq m \leq n$ holds. The two angles are real-valued with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$.

See also:

[Ynm_c](#) (page 552), [Znm](#) (page 552)

References

[R360] (page 1865), [R361] (page 1865), [R362] (page 1865), [R363] (page 1865)

Examples

```
>>> from sympy import Ynm, Symbol
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)
```

Several symmetries are known, for the order

```
>>> from sympy import Ynm, Symbol
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, -m, theta, phi)
(-1)**m*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

as well as for the angles

```
>>> from sympy import Ynm, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)
```

```
>>> Ynm(n, m, theta, -phi)
exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions

```
>>> simplify(Ynm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(-I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(1, 0, theta, phi).expand(func=True))
sqrt(3)*cos(theta)/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, 1, theta, phi).expand(func=True))
-sqrt(6)*exp(I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, -2, theta, phi).expand(func=True))
sqrt(30)*exp(-2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, -1, theta, phi).expand(func=True))
sqrt(30)*exp(-I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 0, theta, phi).expand(func=True))
sqrt(5)*(3*cos(theta)**2 - 1)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, 1, theta, phi).expand(func=True))
-sqrt(30)*exp(I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 2, theta, phi).expand(func=True))
sqrt(30)*exp(2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

We can differentiate the functions with respect to both angles

```
>>> from sympy import Ynm, Symbol, diff
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + sqrt((-m + n)*(m + n + 1))*exp(-I*phi)*Ynm(n,
↪ m + 1, theta, phi)
```

```
>>> diff(Ynm(n, m, theta, phi), phi)
I*m*Ynm(n, m, theta, phi)
```

Further we can compute the complex conjugation

```
>>> from sympy import Ynm, Symbol, conjugate
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> conjugate(Ynm(n, m, theta, phi))
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

To get back the well known expressions in spherical coordinates we use full expansion

```
>>> from sympy import Ynm, Symbol, expand_func
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> expand_func(Ynm(n, m, theta, phi))
sqrt((2*n + 1)*factorial(-m + n)/factorial(m + n))*exp(I*m*phi)*assoc_legendre(n,
↪ m, cos(theta))/(2*sqrt(pi))
```

`sympy.functions.special.spherical_harmonics.Ynm_c(n, m, theta, phi)`
Conjugate spherical harmonics defined as

$$\overline{Y_n^m(\theta, \varphi)} := (-1)^m Y_n^{-m}(\theta, \varphi)$$

See also:

[Ynm](#) (page 550), [Znm](#) (page 552)

References

[R364] (page 1865), [R365] (page 1865), [R366] (page 1865)

class `sympy.functions.special.spherical_harmonics.Znm`
Real spherical harmonics defined as

$$Z_n^m(\theta, \varphi) := \begin{cases} \frac{Y_n^m(\theta, \varphi) + \overline{Y_n^m(\theta, \varphi)}}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - \overline{Y_n^m(\theta, \varphi)}}{i\sqrt{2}} & m < 0 \end{cases}$$

which gives in simplified form

$$Z_n^m(\theta, \varphi) = \begin{cases} \frac{Y_n^m(\theta, \varphi) + (-1)^m Y_n^{-m}(\theta, \varphi)}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - (-1)^m Y_n^{-m}(\theta, \varphi)}{i\sqrt{2}} & m < 0 \end{cases}$$

See also:

[Ynm](#) (page 550), [Ynm_c](#) (page 552)

References

[R367] (page 1865), [R368] (page 1865), [R369] (page 1865)

Tensor Functions

`sympy.functions.special.tensor_functions.Eijk(*args, **kwargs)`

Represent the Levi-Civita symbol.

This is just compatibility wrapper to `LeviCivita()`.

See also:

[LeviCivita](#) (page 553)

`sympy.functions.special.tensor_functions.eval_levicivita(*args)`

Evaluate Levi-Civita symbol.

class `sympy.functions.special.tensor_functions.LeviCivita`

Represent the Levi-Civita symbol.

For even permutations of indices it returns 1, for odd permutations -1, and for everything else (a repeated index) it returns 0.

Thus it represents an alternating pseudotensor.

See also:

[Eijk](#) (page 553)

Examples

```
>>> from sympy import LeviCivita
>>> from sympy.abc import i, j, k
>>> LeviCivita(1, 2, 3)
1
>>> LeviCivita(1, 3, 2)
-1
>>> LeviCivita(1, 2, 2)
0
>>> LeviCivita(i, j, k)
LeviCivita(i, j, k)
>>> LeviCivita(i, j, i)
0
```

class `sympy.functions.special.tensor_functions.KroneckerDelta`

The discrete, or Kronecker, delta function.

A function that takes in two integers i and j . It returns 0 if i and j are not equal or it returns 1 if i and j are equal.

Parameters **i** : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

See also:

`eval` (page 554), `sympy.functions.special.delta_functions.DiracDelta` (page 475)

References

[R370] (page 1865)

Examples

A simple example with integer indices:

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

classmethod `eval(i, j)`

Evaluates the discrete delta function.

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

indirect doctest

indices_contain_equal_information

Returns True if indices are either both above or below fermi.

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

is_above_fermi

True if Delta can be non-zero above fermi

See also:

[is_below_fermi](#) (page 555), [is_only_below_fermi](#) (page 556),
[is_only_above_fermi](#) (page 556)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

is_below_fermi

True if Delta can be non-zero below fermi

See also:

[is_above_fermi](#) (page 555), [is_only_above_fermi](#) (page 556),
[is_only_below_fermi](#) (page 556)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
```

(continues on next page)

(continued from previous page)

```
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

is_only_above_fermi

True if Delta is restricted to above fermi

See also:

[is_above_fermi](#) (page 555), [is_below_fermi](#) (page 555), [is_only_below_fermi](#) (page 556)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

is_only_below_fermi

True if Delta is restricted to below fermi

See also:

[is_above_fermi](#) (page 555), [is_below_fermi](#) (page 555), [is_only_above_fermi](#) (page 556)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

killable_index

Returns the index which is preferred to substitute in the final expression.

The index to substitute is the index with less information regarding fermi level. If indices contain same information, 'a' is preferred before 'b'.

See also:

[preferred_index](#) (page 557)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

preferred_index

Returns the index which is preferred to keep in the final expression.

The preferred index is the index with more information regarding fermi level. If indices contain same information, 'a' is preferred before 'b'.

See also:

[killable_index](#) (page 556)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

5.10 Geometry Module

5.10.1 Introduction

The geometry module for SymPy allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include

asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

5.10.2 Available Entities

The following entities are currently available in the geometry module:

- [Point](#) (page 567)
- [Line](#) (page 589), [Segment](#) (page 594), [Ray](#) (page 591)
- [Ellipse](#) (page 612), [Circle](#) (page 624)
- [Polygon](#) (page 627), [RegularPolygon](#) (page 634), [Triangle](#) (page 641)

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- `intersection(entity1, entity2)`
- `are_similar(entity1, entity2)`
- `convex_hull(points)`

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

5.10.3 Example Usage

The following Python session gives one an idea of how to work with some of the geometry module.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Triangle(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
[Point2D(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l) # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l) # is l tangent to c?
False
```

(continues on next page)

(continued from previous page)

```
>>> intersection(c, l)
[Point2D(-5*sqrt(2)/2, -5*sqrt(2)/2), Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
```

5.10.4 Intersection of medians

```
>>> from sympy import symbols
>>> from sympy.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(3*a/2, b/2))

>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point2D(a, b/3)]
```

5.10.5 An in-depth example: Pappus' Hexagon Theorem

From Wikipedia ([WikiPappus] (page 1865)):

Given one set of collinear points A, B, C , and another set of collinear points a, b, c , then the intersection points X, Y, Z of line pairs Ab and aB , Ac and aC , Bc and bC are collinear.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>>
>>> l1 = Line(Point(0, 0), Point(5, 6))
>>> l2 = Line(Point(0, 0), Point(2, -2))
>>>
>>> def subs_point(l, val):
...     """Take an arbitrary point and make it a fixed point."""
...     t = Symbol('t', real=True)
...     ap = l.arbitrary_point()
...     return Point(ap.x.subs(t, val), ap.y.subs(t, val))
...
>>> p11 = subs_point(l1, 5)
>>> p12 = subs_point(l1, 6)
>>> p13 = subs_point(l1, 11)
>>>
>>> p21 = subs_point(l2, -1)
>>> p22 = subs_point(l2, 2)
>>> p23 = subs_point(l2, 13)
>>>
```

(continues on next page)

(continued from previous page)

```

>>> ll1 = Line(p11, p22)
>>> ll2 = Line(p11, p23)
>>> ll3 = Line(p12, p21)
>>> ll4 = Line(p12, p23)
>>> ll5 = Line(p13, p21)
>>> ll6 = Line(p13, p22)
>>>
>>> pp1 = intersection(ll1, ll3)[0]
>>> pp2 = intersection(ll2, ll5)[0]
>>> pp3 = intersection(ll4, ll6)[0]
>>>
>>> Point.is_collinear(pp1, pp2, pp3)
True

```

References

5.10.6 Miscellaneous Notes

- The area property of Polygon and Triangle may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the abs function.
- Although Polygon can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).
- Since SymPy is still in its infancy some things may not simplify properly and hence some things that should return True (e.g., Point.is_collinear) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

5.10.7 Future Work

Truth Setting Expressions

When one deals with symbolic entities, it often happens that an assertion cannot be guaranteed. For example, consider the following code:

```

>>> from sympy import *
>>> from sympy.geometry import *
>>> x,y,z = map(Symbol, 'xyz')
>>> p1,p2,p3 = Point(x, y), Point(y, z), Point(2*x*y, y)
>>> Point.is_collinear(p1, p2, p3)
False

```

Even though the result is currently False, this is not always true. If the quantity $z - y - 2 * y * z + 2 * y * *2 == 0$ then the points will be collinear. It would be really nice to inform the user of this because such a quantity may be useful to a user for further calculation and, at the very least, being nice to know. This could be potentially done by returning an object (e.g., GeometryResult) that the user could use. This actually would not involve an extensive amount of work.

Three Dimensions and Beyond

Currently a limited subset of the geometry module has been extended to three dimensions, but it certainly would be a good addition to extend more. This would probably involve a fair amount of work since many of the algorithms used are specific to two dimensions.

Geometry Visualization

The plotting module is capable of plotting geometric entities. See [Plotting Geometric Entities](#) (page 1075) in the plotting module entry.

Submodules

Entities

class `sympy.geometry.entity.GeometryEntity`

The base class for all geometrical entities.

This class doesn't represent any particular geometric entity, it only provides the implementation of some methods common to all subclasses.

ambient_dimension

What is the dimension of the space that the object is contained in?

bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

encloses(o)

Return True if o is inside (not on or outside) the boundaries of self.

The object will be decomposed into Points and individual Entities need only define an `encloses_point` method for their class.

See also:

[sympy.geometry.ellipse.Ellipse.encloses_point](#) (page 615), [sympy.geometry.polygon.Polygon.encloses_point](#) (page 630)

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t2 = Polygon(*RegularPolygon(Point(0, 0), 2, 3).vertices)
>>> t2.encloses(t)
True
>>> t.encloses(t2)
False
```

intersection(o)

Returns a list of all of the intersections of self with o.

See also:

[sympy.geometry.util.intersection](#) (page 564)

Notes

An entity is not required to implement this method.

If two different types of entities can intersect, the item with higher index in `ordering_of_classes` should implement intersections with anything having a lower index.

is_similar(other)

Is this geometrical entity similar to another geometrical entity?

Two entities are similar if a uniform scaling (enlarging or shrinking) of one of the entities will allow one to obtain the other.

See also:

[scale](#) (page 563)

Notes

This method is not intended to be used directly but rather through the *are_similar* function found in `util.py`. An entity is not required to implement this method. If two different types of entities can be similar, it is only required that one of them be able to determine this.

parameter_value(other, t)

Return the parameter corresponding to the given point. Evaluating an arbitrary point of the entity at this parameter value will return the given point.

Examples

```
>>> from sympy import Line, Point
>>> from sympy.abc import t
>>> a = Point(0, 0)
>>> b = Point(2, 2)
>>> Line(a, b).parameter_value((1, 1), t)
{t: 1/2}
>>> Line(a, b).arbitrary_point(t).subs(_
Point2D(1, 1)
```

reflect(line)

Reflects an object across a line.

Parameters line: Line

Examples

```
>>> from sympy import pi, sqrt, Line, RegularPolygon
>>> l = Line((0, pi), slope=sqrt(2))
>>> pent = RegularPolygon((1, 2), 1, 5)
>>> rpent = pent.reflect(l)
>>> rpent
RegularPolygon(Point2D(-2*sqrt(2)*pi/3 - 1/3 + 4*sqrt(2)/3, 2/3 + 2*sqrt(2)/3,
+ 2*pi/3), -1, 5, -atan(2*sqrt(2)) + 3*pi/5)
```



```
>>> from sympy import pi, Line, Circle, Point
>>> l = Line((0, pi), slope=1)
>>> circ = Circle(Point(0, 0), 5)
>>> rcirc = circ.reflect(l)
>>> rcirc
Circle(Point2D(-pi, pi), -5)
```

rotate(angle, pt=None)

Rotate angle radians counterclockwise about Point pt.

The default pt is the origin, Point(0, 0)

See also:

[scale](#) (page 563), [translate](#) (page 563)

Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2, -1/2))
```

scale(x=1, y=1, pt=None)

Scale the object by multiplying the x,y-coordinates by x and y.

If pt is given, the scaling is done relative to that point; the object is shifted by -pt, scaled, and shifted by pt.

See also:

[rotate](#) (page 563), [translate](#) (page 563)

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.scale(2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)/2), Point2D(-1, -sqrt(3)/2))
>>> t.scale(2, 2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)), Point2D(-1, -sqrt(3)))
```

translate(x=0, y=0)

Shift the object by adding to the x,y-coordinates the values x and y.

See also:

[rotate](#) (page 563), [scale](#) (page 563)

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.translate(2)
Triangle(Point2D(3, 0), Point2D(3/2, sqrt(3)/2), Point2D(3/2, -sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point2D(3, 2), Point2D(3/2, sqrt(3)/2 + 2),
        Point2D(3/2, -sqrt(3)/2 + 2))
```

Utils

`sympy.geometry.util.intersection(*entities, **kwargs)`

The intersection of a collection of GeometryEntity instances.

Parameters **entities** : sequence of GeometryEntity

pairwise (keyword argument) : Can be either True or False

Returns **intersection** : list of GeometryEntity

Raises **NotImplementedError**

When unable to calculate intersection.

See also:

[sympy.geometry.entity.GeometryEntity.intersection](#) (page 561)

Notes

The intersection of any geometrical entity with itself should return a list with one item: the entity in question. An intersection requires two or more entities. If only a single entity is given then the function will return an empty list. It is possible for *intersection* to miss intersections that one knows exists because the required quantities were not fully simplified internally. Reals should be converted to Rationals, e.g. `Rational(str(real_num))` or else failures due to floating point issues may result.

Case 1: When the keyword argument 'pairwise' is False (default value): In this case, the function returns a list of intersections common to all entities.

Case 2: When the keyword argument 'pairwise' is True: In this case, the functions returns a list intersections that occur between any pair of entities.

Examples

```
>>> from sympy.geometry import Ray, Circle, intersection
>>> c = Circle((0, 1), 1)
>>> intersection(c, c.center)
[]
>>> right = Ray((0, 0), (1, 0))
>>> up = Ray((0, 0), (0, 1))
>>> intersection(c, right, up)
```

(continues on next page)

(continued from previous page)

```
[Point2D(0, 0)]
>>> intersection(c, right, up, pairwise=True)
[Point2D(0, 0), Point2D(0, 2)]
>>> left = Ray((1, 0), (0, 0))
>>> intersection(right, left)
[Segment2D(Point2D(0, 0), Point2D(1, 0))]
```

`sympy.geometry.util.convex_hull(*args, **kwargs)`

The convex hull surrounding the Points contained in the list of entities.

Parameters `args` : a collection of Points, Segments and/or Polygons

Returns `convex_hull` : Polygon if `polygon` is True else as a tuple (U, L) where L and U are the lower and upper hulls, respectively.

See also:

`sympy.geometry.point.Point` (page 567), `sympy.geometry.polygon.Polygon` (page 627)

Notes

This can only be performed on a set of points whose coordinates can be ordered on the number line.

References

[1] http://en.wikipedia.org/wiki/Graham_scan

[2] Andrew's Monotone Chain Algorithm (A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", 1979) http://geomalgorithms.com/a10-_hull-1.html

Examples

```
>>> from sympy.geometry import Point, convex_hull
>>> points = [(1, 1), (1, 2), (3, 1), (-5, 2), (15, 4)]
>>> convex_hull(*points)
Polygon(Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4))
>>> convex_hull(*points, **dict(polygon=False))
([Point2D(-5, 2), Point2D(15, 4)],
 [Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4)])
```

`sympy.geometry.util.are_similar(e1, e2)`

Are two geometrical entities similar.

Can one geometrical entity be uniformly scaled to the other?

Parameters `e1` : GeometryEntity

`e2` : GeometryEntity

Returns `are_similar` : boolean

Raises `GeometryError`

When $e1$ and $e2$ cannot be compared.

See also:

[sympy.geometry.entity.GeometryEntity.is_similar](#) (page 562)

Notes

If the two objects are equal then they are similar.

Examples

```
>>> from sympy import Point, Circle, Triangle, are_similar
>>> c1, c2 = Circle(Point(0, 0), 4), Circle(Point(1, 4), 3)
>>> t1 = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
>>> t2 = Triangle(Point(0, 0), Point(2, 0), Point(0, 2))
>>> t3 = Triangle(Point(0, 0), Point(3, 0), Point(0, 1))
>>> are_similar(t1, t2)
True
>>> are_similar(t1, t3)
False
```

`sympy.geometry.util.centroid(*args)`

Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons. The centroid is the weighted average of the individual centroid where the weights are the lengths (of segments) or areas (of polygons). Overlapping regions will add to the weight of that region.

If there are no objects (or a mixture of objects) then None is returned.

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Segment](#) (page 594), [sympy.geometry.polygon.Polygon](#) (page 627)

Examples

```
>>> from sympy import Point, Segment, Polygon
>>> from sympy.geometry.util import centroid
>>> p = Polygon((0, 0), (10, 0), (10, 10))
>>> q = p.translate(0, 20)
>>> p.centroid, q.centroid
(Point2D(20/3, 10/3), Point2D(20/3, 70/3))
>>> centroid(p, q)
Point2D(20/3, 40/3)
>>> p, q = Segment((0, 0), (2, 0)), Segment((0, 0), (2, 2))
>>> centroid(p, q)
Point2D(1, -sqrt(2) + 2)
>>> centroid(Point(0, 0), Point(2, 0))
Point2D(1, 0)
```

Stacking 3 polygons on top of each other effectively triples the weight of that polygon:

```
>>> p = Polygon((0, 0), (1, 0), (1, 1), (0, 1))
>>> q = Polygon((1, 0), (3, 0), (3, 1), (1, 1))
>>> centroid(p, q)
```

(continues on next page)

(continued from previous page)

```
Point2D(3/2, 1/2)
>>> centroid(p, p, p, q) # centroid x-coord shifts left
Point2D(11/10, 1/2)
```

Stacking the squares vertically above and below p has the same effect:

```
>>> centroid(p, p.translate(0, 1), p.translate(0, -1), q)
Point2D(11/10, 1/2)
```

Points

class `sympy.geometry.point.Point`

A point in a n -dimensional Euclidean space.

Parameters `coords` : sequence of n -coordinate values. In the special case where $n=2$ or 3 , a `Point2D` or `Point3D` will be created as appropriate.

evaluate : if `True` (default), all floats are turn into exact types.

dim : number of coordinates the point should have. If coordinates are unspecified, they are padded with zeros.

on_morph : indicates what should happen when the number of coordinates of a point need to be changed by adding or removing zeros. Possible values are `'warn'`, `'error'`, or `ignore` (default). No warning or error is given when `*args` is empty and `dim` is given. An error is always raised when trying to remove nonzero coordinates.

Raises **TypeError** : When instantiating with anything but a `Point` or sequence

ValueError : when instantiating with a sequence with length < 2 or when trying to reduce dimensions if keyword `on_morph = 'error'` is set.

See also:

`sympy.geometry.line.Segment` (page 594) Connects two Points

Examples

```
>>> from sympy.geometry import Point
>>> from sympy.abc import x
>>> Point(1, 2, 3)
Point3D(1, 2, 3)
>>> Point([1, 2])
Point2D(1, 2)
>>> Point(0, x)
Point2D(0, x)
>>> Point(dim=4)
Point(0, 0, 0, 0)
```

Floats are automatically converted to Rational unless the `evaluate` flag is `False`:

```
>>> Point(0.5, 0.25)
Point2D(1/2, 1/4)
>>> Point(0.5, 0.25, evaluate=False)
Point2D(0.5, 0.25)
```

Attributes

length	
origin: A <i>Point</i> representing the origin of the	appropriately-dimensioned space.

static affine_rank(*args)

The affine rank of a set of points is the dimension of the smallest affine space containing all the points. For example, if the points lie on a line (and are not all the same) their affine rank is 1. If the points lie on a plane but not a line, their affine rank is 2. By convention, the empty set has affine rank -1.

ambient_dimension

Number of components this point has.

classmethod are_coplanar(*points)

Return True if there exists a plane in which all the points lie. A trivial True value is returned if $\text{len}(\text{points}) < 3$ or all Points are 2-dimensional.

Parameters A set of points

Returns boolean

Raises ValueError : if less than 3 unique points are given

Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 2)
>>> p2 = Point3D(2, 7, 2)
>>> p3 = Point3D(0, 0, 2)
>>> p4 = Point3D(1, 1, 2)
>>> Point3D.are_coplanar(p1, p2, p3, p4)
True
>>> p5 = Point3D(0, 1, 3)
>>> Point3D.are_coplanar(p1, p2, p3, p5)
False
```

canberra_distance(p)

The Canberra Distance from self to point p.

Returns the weighted sum of horizontal and vertical distances to point p.

Parameters **p** : Point

Returns canberra_distance : The weighted sum of horizontal and vertical distances to point p. The weight used is the sum of absolute values of the coordinates.

Raises ValueError when both vectors are zero.

See also:

[sympy.geometry.point.Point.distance](#) (page 569)

Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(3, 3)
>>> p1.canberra_distance(p2)
1
>>> p1, p2 = Point(0, 0), Point(3, 3)
>>> p1.canberra_distance(p2)
2
```

distance(p)

The Euclidean distance from self to point p.

Parameters **p** : Point

Returns **distance** : number or symbolic expression.

See also:

[sympy.geometry.line.Segment.length](#) (page 596), [sympy.geometry.point.Point.taxicab_distance](#) (page 573)

Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> p1.distance(p2)
5
```

```
>>> from sympy.abc import x, y
>>> p3 = Point(x, y)
>>> p3.distance(Point(0, 0))
sqrt(x**2 + y**2)
```

dot(p)

Return dot product of self with another Point.

equals(other)

Returns whether the coordinates of self and other agree.

evalf(prec=None, **options)

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

Parameters **prec** : int

Returns **point** : Point

Examples

```
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point2D(1/2, 3/2)
>>> p1.evalf()
Point2D(0.5, 1.5)
```

intersection(other)

The intersection between this point and another GeometryEntity.

Parameters other : Point

Returns intersection : list of Points

Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

Examples

```
>>> from sympy import Point
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point2D(0, 0)]
```

is_collinear(*args)

Returns *True* if there exists a line that contains *self* and *points*. Returns *False* otherwise. A trivially True value is returned if no points are given.

Parameters args : sequence of Points

Returns is_collinear : boolean

See also:

[sympy.geometry.line.Line](#) (page 589)

Examples

```
>>> from sympy import Point
>>> from sympy.abc import x
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4, p5 = Point(2, 2), Point(x, x), Point(1, 2)
>>> Point.is_collinear(p1, p2, p3, p4)
True
>>> Point.is_collinear(p1, p2, p3, p5)
False
```


is_concyclic(*args)

Do *self* and the given sequence of points lie in a circle?

Returns True if the set of points are concyclic and False otherwise. A trivial value of True is returned if there are fewer than 2 other points.

Parameters *args* : sequence of Points

Returns *is_concyclic* : boolean

Examples

```
>>> from sympy import Point
```

Define 4 points that are on the unit circle:

```
>>> p1, p2, p3, p4 = Point(1, 0), (0, 1), (-1, 0), (0, -1)
```

```
>>> p1.is_concyclic() == p1.is_concyclic(p2, p3, p4) == True
True
```

Define a point not on that circle:

```
>>> p = Point(1, 1)
```

```
>>> p.is_concyclic(p1, p2, p3)
False
```

is_nonzero

True if any coordinate is nonzero, False if every coordinate is zero, and None if it cannot be determined.

is_scalar_multiple(p)

Returns whether each coordinate of *self* is a scalar multiple of the corresponding coordinate in point *p*.

is_zero

True if every coordinate is zero, False if any coordinate is not zero, and None if it cannot be determined.

length

Treating a Point as a Line, this returns 0 for the length of a Point.

Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.length
0
```

midpoint(p)

The midpoint between self and point *p*.

Parameters *p* : Point

Returns *midpoint* : Point

See also:

[sympy.geometry.line.Segment.midpoint](#) (page 596)

Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(13, 5)
>>> p1.midpoint(p2)
Point2D(7, 3)
```

n(prec=None, **options)

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

Parameters *prec* : int

Returns *point* : Point

Examples

```
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point2D(1/2, 3/2)
>>> p1.evalf()
Point2D(0.5, 1.5)
```

origin

A point of all zeros of the same ambient dimension as the current point

orthogonal_direction

Returns a non-zero point that is orthogonal to the line containing *self* and the origin.

Examples

```
>>> from sympy.geometry import Line, Point
>>> a = Point(1, 2, 3)
>>> a.orthogonal_direction
Point3D(-2, 1, 0)
>>> b = _
>>> Line(b, b.origin).is_perpendicular(Line(a, a.origin))
True
```

static project(*a*, *b*)

Project the point *a* onto the line between the origin and point *b* along the normal direction.

Parameters *a* : Point

b : Point

Returns *p* : Point

See also:

[sympy.geometry.line.LinearEntity.projection](#) (page 587)

Examples

```
>>> from sympy.geometry import Line, Point
>>> a = Point(1, 2)
>>> b = Point(2, 5)
>>> z = a.origin
>>> p = Point.project(a, b)
>>> Line(p, a).is_perpendicular(Line(p, b))
True
>>> Point.is_collinear(z, p, b)
True
```

taxicab_distance(p)

The Taxicab Distance from self to point p.

Returns the sum of the horizontal and vertical distances to point p.

Parameters *p* : Point

Returns **taxicab_distance** : The sum of the horizontal and vertical distances to point p.

See also:

[sympy.geometry.point.Point.distance](#) (page 569)

Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> p1.taxicab_distance(p2)
7
```

unit

Return the Point that is in the same direction as *self* and a distance of 1 from the origin

class `sympy.geometry.point.Point2D`

A point in a 2-dimensional Euclidean space.

Parameters **coords** : sequence of 2 coordinate values.

Raises **TypeError**

When trying to add or subtract points with different dimensions.
When trying to create a point with more than two dimensions. When *intersection* is called with object other than a Point.

See also:

[sympy.geometry.line.Segment](#) (page 594) Connects two Points

Examples

```
>>> from sympy.geometry import Point2D
>>> from sympy.abc import x
>>> Point2D(1, 2)
Point2D(1, 2)
>>> Point2D([1, 2])
Point2D(1, 2)
>>> Point2D(0, x)
Point2D(0, x)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point2D(0.5, 0.25)
Point2D(1/2, 1/4)
>>> Point2D(0.5, 0.25, evaluate=False)
Point2D(0.5, 0.25)
```

Attributes

x	
y	
length	

bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

rotate(angle, pt=None)

Rotate angle radians counterclockwise about Point pt.

See also:

[rotate](#) (page 574), [scale](#) (page 574)

Examples

```
>>> from sympy import Point2D, pi
>>> t = Point2D(1, 0)
>>> t.rotate(pi/2)
Point2D(0, 1)
>>> t.rotate(pi/2, (2, 0))
Point2D(2, -1)
```

scale(x=1, y=1, pt=None)

Scale the coordinates of the Point by multiplying by x and y after subtracting pt - default is (0, 0) - and then adding pt back again (i.e. pt is the point of reference for the scaling).

See also:

[rotate](#) (page 574), [translate](#) (page 575)

Examples

```
>>> from sympy import Point2D
>>> t = Point2D(1, 1)
>>> t.scale(2)
Point2D(2, 1)
>>> t.scale(2, 2)
Point2D(2, 2)
```

transform(matrix)

Return the point after applying the transformation described by the 3x3 Matrix, matrix.

See also:

`geometry.entity.rotate`, `geometry.entity.scale`, `geometry.entity.translate`

translate(x=0, y=0)

Shift the Point by adding x and y to the coordinates of the Point.

See also:

`rotate` (page 574), `scale` (page 574)

Examples

```
>>> from sympy import Point2D
>>> t = Point2D(0, 1)
>>> t.translate(2)
Point2D(2, 1)
>>> t.translate(2, 2)
Point2D(2, 3)
>>> t + Point2D(2, 2)
Point2D(2, 3)
```

x

Returns the X coordinate of the Point.

Examples

```
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.x
0
```

y

Returns the Y coordinate of the Point.

Examples

```
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.y
1
```

class `sympy.geometry.point.Point3D`

A point in a 3-dimensional Euclidean space.

Parameters `coords` : sequence of 3 coordinate values.

Raises `TypeError`

When trying to add or subtract points with different dimensions.

When *intersection* is called with object other than a Point.

Examples

```
>>> from sympy import Point3D
>>> from sympy.abc import x
>>> Point3D(1, 2, 3)
Point3D(1, 2, 3)
>>> Point3D([1, 2, 3])
Point3D(1, 2, 3)
>>> Point3D(0, x, 3)
Point3D(0, x, 3)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point3D(0.5, 0.25, 2)
Point3D(1/2, 1/4, 2)
>>> Point3D(0.5, 0.25, 3, evaluate=False)
Point3D(0.5, 0.25, 3)
```

Attributes

x	
y	
z	
length	

static `are_collinear(*points)`

Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

Parameters `points` : sequence of Point

Returns `are_collinear` : boolean

See also:

[sympy.geometry.line.Line3D](#) (page 604)

Examples

```
>>> from sympy import Point3D, Matrix
>>> from sympy.abc import x
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> p3, p4, p5 = Point3D(2, 2, 2), Point3D(x, x, x), Point3D(1, 2, 6)
>>> Point3D.are_collinear(p1, p2, p3, p4)
True
>>> Point3D.are_collinear(p1, p2, p3, p5)
False
```

direction_cosine(point)

Gives the direction cosine between 2 points

Parameters **p** : Point3D

Returns list

Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_cosine(Point3D(2, 3, 5))
[sqrt(6)/6, sqrt(6)/6, sqrt(6)/3]
```

direction_ratio(point)

Gives the direction ratio between 2 points

Parameters **p** : Point3D

Returns list

Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_ratio(Point3D(2, 3, 5))
[1, 1, 2]
```

intersection(other)

The intersection between this point and another point.

Parameters **other** : Point

Returns **intersection** : list of Points

Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

Examples

```
>>> from sympy import Point3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point3D(0, 0, 0)]
```

scale(x=1, y=1, z=1, pt=None)

Scale the coordinates of the Point by multiplying by x and y after subtracting pt - default is (0, 0) - and then adding pt back again (i.e. pt is the point of reference for the scaling).

See also:

[translate](#) (page 578)

Examples

```
>>> from sympy import Point3D
>>> t = Point3D(1, 1, 1)
>>> t.scale(2)
Point3D(2, 1, 1)
>>> t.scale(2, 2)
Point3D(2, 2, 1)
```

transform(matrix)

Return the point after applying the transformation described by the 4x4 Matrix, matrix.

See also:

[geometry.entity.rotate](#), [geometry.entity.scale](#), [geometry.entity.translate](#)

translate(x=0, y=0, z=0)

Shift the Point by adding x and y to the coordinates of the Point.

See also:

[rotate](#), [scale](#) (page 578)

Examples

```
>>> from sympy import Point3D
>>> t = Point3D(0, 1, 1)
>>> t.translate(2)
Point3D(2, 1, 1)
>>> t.translate(2, 2)
Point3D(2, 3, 1)
>>> t + Point3D(2, 2, 2)
Point3D(2, 3, 3)
```

x

Returns the X coordinate of the Point.

Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 3)
>>> p.x
0
```

y

Returns the Y coordinate of the Point.

Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.y
1
```

z

Returns the Z coordinate of the Point.

Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 1)
>>> p.z
1
```

Lines

class sympy.geometry.line.LinearEntity

A base class for all linear entities (Line, Ray and Segment) in n-dimensional Euclidean space.

See also:

[sympy.geometry.entity.GeometryEntity](#) (page 561)

Notes

This is an abstract class and is not meant to be instantiated.

Attributes

ambient_dimension	
direction	
length	
p1	
p2	
points	

ambient_dimension

A property method that returns the dimension of LinearEntity object.

Parameters **p1** : LinearEntity

Returns **dimension** : integer

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> l1 = Line(p1, p2)
>>> l1.ambient_dimension
2
```

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 1)
>>> l1 = Line(p1, p2)
>>> l1.ambient_dimension
3
```

angle_between(l2)

Return the non-reflex angle formed by rays emanating from the origin with directions the same as the direction vectors of the linear entities.

Parameters **l1** : LinearEntity

l2 : LinearEntity

Returns **angle** : angle in radians

See also:

[is_perpendicular](#) (page 584), [Ray2D.closing_angle](#) (page 601)

Notes

From the dot product of vectors v_1 and v_2 it is known that:

$$\text{dot}(v_1, v_2) = |v_1| * |v_2| * \cos(A)$$

where A is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

Examples

```
>>> from sympy import Point, Line, pi
>>> e = Line((0, 0), (1, 0))
>>> ne = Line((0, 0), (1, 1))
>>> sw = Line((1, 1), (0, 0))
>>> ne.angle_between(e)
pi/4
>>> sw.angle_between(e)
3*pi/4
```

To obtain the non-obtuse angle at the intersection of lines, use the `smallest_angle_between` method:

```
>>> sw.smallest_angle_between(e)
pi/4
```

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.angle_between(l2)
acos(-sqrt(2)/3)
>>> l1.smallest_angle_between(l2)
acos(sqrt(2)/3)
```

arbitrary_point(parameter='t')

A parameterized point on the Line.

Parameters `parameter` : str, optional

The name of the parameter which will be used for the parametric point. The default value is 't'. When this parameter is 0, the first point used to define the line will be returned, and when it is 1 the second point will be returned.

Returns `point` : Point

Raises `ValueError`

When parameter already appears in the Line's definition.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.arbitrary_point()
Point2D(4*t + 1, 3*t)
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.arbitrary_point()
Point3D(4*t + 1, 3*t, t)
```

static `are_concurrent`(*lines)

Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

Parameters `lines` : a sequence of linear entities.

Returns `True` : if the set of linear entities intersect in one point

`False` : otherwise.

See also:

[sympy.geometry.util.intersection](#) (page 564)

Examples

```
>>> from sympy import Point, Line, Line3D
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> p3, p4 = Point(-2, -2), Point(0, 2)
>>> l1, l2, l3 = Line(p1, p2), Line(p1, p3), Line(p1, p4)
>>> Line.are_concurrent(l1, l2, l3)
True
>>> l4 = Line(p2, p3)
>>> Line.are_concurrent(l2, l3, l4)
False
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 5, 2)
>>> p3, p4 = Point3D(-2, -2, -2), Point3D(0, 2, 1)
>>> l1, l2, l3 = Line3D(p1, p2), Line3D(p1, p3), Line3D(p1, p4)
>>> Line3D.are_concurrent(l1, l2, l3)
True
>>> l4 = Line3D(p2, p3)
>>> Line3D.are_concurrent(l2, l3, l4)
False
```

contains(other)

Subclasses should implement this method and should return True if other is on the boundaries of self; False if not on the boundaries of self; None if a determination cannot be made.

direction

The direction vector of the LinearEntity.

Returns **p** : a Point; the ray from the origin to this point is the direction of *self*

See also:

[sympy.geometry.point.Point.unit](#) (page 573)

Examples

```
>>> from sympy.geometry import Line
>>> a, b = (1, 1), (1, 3)
>>> Line(a, b).direction
Point2D(0, 2)
>>> Line(b, a).direction
Point2D(0, -2)
```

This can be reported so the distance from the origin is 1:

```
>>> Line(b, a).direction.unit
Point2D(0, -1)
```

intersection(other)

The intersection with another geometrical entity.

Parameters **o** : Point or LinearEntity

Returns **intersection** : list of geometrical entities

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Line, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(7, 7)
>>> l1 = Line(p1, p2)
>>> l1.intersection(p3)
[Point2D(7, 7)]
>>> p4, p5 = Point(5, 0), Point(0, 3)
>>> l2 = Line(p4, p5)
>>> l1.intersection(l2)
[Point2D(15/8, 15/8)]
>>> p6, p7 = Point(0, 5), Point(2, 6)
>>> s1 = Segment(p6, p7)
>>> l1.intersection(s1)
[]
>>> from sympy import Point3D, Line3D, Segment3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(7, 7, 7)
>>> l1 = Line3D(p1, p2)
>>> l1.intersection(p3)
[Point3D(7, 7, 7)]
>>> l1 = Line3D(Point3D(4, 19, 12), Point3D(5, 25, 17))
>>> l2 = Line3D(Point3D(-3, -15, -19), direction_ratio=[2, 8, 8])
>>> l1.intersection(l2)
[Point3D(1, 1, -3)]
>>> p6, p7 = Point3D(0, 5, 2), Point3D(2, 6, 3)
>>> s1 = Segment3D(p6, p7)
>>> l1.intersection(s1)
[]
```

is_parallel(l2)

Are two linear entities parallel?

Parameters **l1** : LinearEntity

l2 : LinearEntity

Returns **True** : if l1 and l2 are parallel,

False : otherwise.

See also:

[coefficients](#)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4 = Point(3, 4), Point(6, 7)
>>> l1, l2 = Line(p1, p2), Line(p3, p4)
>>> Line.is_parallel(l1, l2)
True
>>> p5 = Point(6, 6)
```

(continues on next page)

(continued from previous page)

```

>>> l3 = Line(p3, p5)
>>> Line.is_parallel(l1, l3)
False
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 4, 5)
>>> p3, p4 = Point3D(2, 1, 1), Point3D(8, 9, 11)
>>> l1, l2 = Line3D(p1, p2), Line3D(p3, p4)
>>> Line3D.is_parallel(l1, l2)
True
>>> p5 = Point3D(6, 6, 6)
>>> l3 = Line3D(p3, p5)
>>> Line3D.is_parallel(l1, l3)
False

```

is_perpendicular(l2)

Are two linear entities perpendicular?

Parameters **l1** : LinearEntity

l2 : LinearEntity

Returns **True** : if l1 and l2 are perpendicular,

False : otherwise.

See also:

coefficients

Examples

```

>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 1)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.is_perpendicular(l2)
True
>>> p4 = Point(5, 3)
>>> l3 = Line(p1, p4)
>>> l1.is_perpendicular(l3)
False
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.is_perpendicular(l2)
False
>>> p4 = Point3D(5, 3, 7)
>>> l3 = Line3D(p1, p4)
>>> l1.is_perpendicular(l3)
False

```

is_similar(other)

Return True if self and other are contained in the same line.

Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 1), Point(3, 4), Point(2, 3)
>>> l1 = Line(p1, p2)
>>> l2 = Line(p1, p3)
>>> l1.is_similar(l2)
True
```

length

The length of the line.

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.length
oo
```

p1

The first defining point of a linear entity.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p1
Point2D(0, 0)
```

p2

The second defining point of a linear entity.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p2
Point2D(5, 3)
```

parallel_line(p)

Create a new Line parallel to this linear entity which passes through the point p .

Parameters p : Point

Returns `line` : `Line`

See also:

[`is\_parallel`](#) (page 583)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
```

`perpendicular_line(p)`

Create a new `Line` perpendicular to this linear entity which passes through the point p .

Parameters `p` : `Point`

Returns `line` : `Line`

See also:

[`is\_perpendicular`](#) (page 584), [`perpendicular\_segment`](#) (page 586)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```


perpendicular_segment(*p*)

Create a perpendicular line segment from *p* to this line.

The endpoints of the segment are *p* and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

Parameters *p* : Point

Returns *segment* : Segment

See also:

[perpendicular_line](#) (page 586)

Notes

Returns *p* itself if *p* is on this linear entity.

Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 2)
>>> l1 = Line(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point(4, 0))
Segment2D(Point2D(4, 0), Point2D(2, 2))
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point3D(4, 0, 0))
Segment3D(Point3D(4, 0, 0), Point3D(4/3, 4/3, 4/3))
```

points

The two points used to define this linear entity.

Returns *points* : tuple of Points

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 11)
>>> l1 = Line(p1, p2)
>>> l1.points
(Point2D(0, 0), Point2D(5, 11))
```

projection(other)

Project a point, line, ray, or segment onto this linear entity.

Parameters other : Point or LinearEntity (Line, Ray, Segment)

Returns projection : Point or LinearEntity (Line, Ray, Segment)

The return type matches the type of the parameter other.

Raises GeometryError

When method is unable to perform projection.

See also:

[sympy.geometry.point.Point](#) (page 567), [perpendicular_line](#) (page 586)

Notes

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point P is projected onto a line L by finding the point on L that is closest to P. This point is the intersection of L and the line perpendicular to L that passes through P.

Examples

```
>>> from sympy import Point, Line, Segment, Rational
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(Rational(1, 2), 0)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point2D(1/4, 1/4)
>>> p4, p5 = Point(10, 0), Point(12, 1)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment2D(Point2D(5, 5), Point2D(13/2, 13/2))
>>> p1, p2, p3 = Point(0, 0, 1), Point(1, 1, 2), Point(2, 0, 1)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point3D(2/3, 2/3, 5/3)
>>> p4, p5 = Point(10, 0, 1), Point(12, 1, 3)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment3D(Point3D(10/3, 10/3, 13/3), Point3D(5, 5, 6))
```

random_point(seed=None)

A random point on a LinearEntity.

Returns point : Point

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Line, Ray, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> line = Line(p1, p2)
>>> r = line.random_point(seed=42) # seed value is optional
>>> r.n(3)
Point2D(-0.72, -0.432)
>>> r in line
True
>>> Ray(p1, p2).random_point(seed=42).n(3)
Point2D(0.72, 0.432)
>>> Segment(p1, p2).random_point(seed=42).n(3)
Point2D(3.2, 1.92)
```

`smallest_angle_between(l1)`

Return the smallest angle formed at the intersection of the lines containing the linear entities.

Parameters `l1` : LinearEntity

`l2` : LinearEntity

Returns `angle` : angle in radians

See also:

[angle_between](#) (page 580), [Ray2D.closing_angle](#) (page 601)

Examples

```
>>> from sympy import Point, Line, pi
>>> p1, p2, p3 = Point(0, 0), Point(0, 4), Point(2, -2)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.smallest_angle_between(l2)
pi/4
```

`class sympy.geometry.line.Line`

An infinite line in space.

A line is declared with two distinct points. A 2D line may be declared with a point and slope and a 3D line may be defined with a point and a direction ratio.

Parameters `p1` : Point

`p2` : Point

`slope` : sympy expression

`direction_ratio` : list

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Line2D](#) (page 599), [sympy.geometry.line.Line3D](#) (page 604)

Notes

Line will automatically subclass to *Line2D* or *Line3D* based on the dimension of *p1*. The *slope* argument is only relevant for *Line2D* and the *direction_ratio* argument is only relevant

for *Line3D*.

Examples

```
>>> from sympy import Point
>>> from sympy.geometry import Line, Segment
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line2D(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword slope:

```
>>> Line(Point(0, 0), slope=0)
Line2D(Point2D(0, 0), Point2D(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

contains(other)

Return True if *other* is on this Line, or False otherwise.

Examples

```
>>> from sympy import Line, Point
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> l = Line(p1, p2)
>>> l.contains(p1)
True
>>> l.contains((0, 1))
True
>>> l.contains((0, 0))
False
>>> a = (0, 0, 0)
>>> b = (1, 1, 1)
```

(continues on next page)

(continued from previous page)

```

>>> c = (2, 2, 2)
>>> l1 = Line(a, b)
>>> l2 = Line(b, a)
>>> l1 == l2
False
>>> l1 in l2
True

```

distance(other)

Finds the shortest distance between a line and a point.

Raises `NotImplementedError` is raised if 'other' is not a Point

Examples

```

>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Line(p1, p2)
>>> s.distance(Point(-1, 1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 1)
>>> s = Line(p1, p2)
>>> s.distance(Point(-1, 1, 1))
2*sqrt(6)/3
>>> s.distance((-1, 1, 1))
2*sqrt(6)/3

```

equals(other)

Returns True if self and other are the same mathematical entities

plot_interval(parameter='t')

The plot interval for the default geometric plot of line. Gives values that will produce a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

Parameters `parameter` : str, optional

Default value is 't'.

Returns `plot_interval` : list (plot interval)

[parameter, lower_bound, upper_bound]

Examples

```

>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]

```

class `sympy.geometry.line.Ray`

A Ray is a semi-line in the space with a source point and a direction.

Parameters **p1** : Point

The source of the Ray

p2 : Point or radian value

This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.

See also:

[sympy.geometry.line.Ray2D](#) (page 600), [sympy.geometry.line.Ray3D](#) (page 605), [sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Line](#) (page 589)

Notes

Ray will automatically subclass to *Ray2D* or *Ray3D* based on the dimension of *p1*.

Examples

```
>>> from sympy import Point, pi
>>> from sympy.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray2D(Point2D(2, 3), Point2D(3, 5))
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
>>> r.source
Point2D(2, 3)
>>> r.xdirection
00
>>> r.ydirection
00
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

Attributes

source	

contains(other)

Is other GeometryEntity contained in this Ray?

Examples

```
>>> from sympy import Ray, Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 4)
>>> r = Ray(p1, p2)
```

(continues on next page)

(continued from previous page)

```

>>> r.contains(p1)
True
>>> r.contains((1, 1))
True
>>> r.contains((1, 3))
False
>>> s = Segment((1, 1), (2, 2))
>>> r.contains(s)
True
>>> s = Segment((1, 2), (2, 5))
>>> r.contains(s)
False
>>> r1 = Ray((2, 2), (3, 3))
>>> r.contains(r1)
True
>>> r1 = Ray((2, 2), (3, 5))
>>> r.contains(r1)
False

```

distance(other)

Finds the shortest distance between the ray and a point.

Raises `NotImplementedError` is raised if 'other' is not a `Point`

Examples

```

>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Ray(p1, p2)
>>> s.distance(Point(-1, -1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 2)
>>> s = Ray(p1, p2)
>>> s
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, 2))
>>> s.distance(Point(-1, -1, 2))
4*sqrt(3)/3
>>> s.distance((-1, -1, 2))
4*sqrt(3)/3

```

equals(other)

Returns True if self and other are the same mathematical entities

plot_interval(parameter='t')

The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

Parameters `parameter` : str, optional

Default value is 't'.

Returns `plot_interval` : list

[parameter, lower_bound, upper_bound]

Examples

```
>>> from sympy import Ray, pi
>>> r = Ray((0, 0), angle=pi/4)
>>> r.plot_interval()
[t, 0, 10]
```

source

The point from which the ray emanates.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.source
Point2D(0, 0)
>>> p1, p2 = Point(0, 0, 0), Point(4, 1, 5)
>>> r1 = Ray(p2, p1)
>>> r1.source
Point3D(4, 1, 5)
```

class sympy.geometry.line.Segment

A line segment in space.

Parameters **p1** : Point

p2 : Point

See also:

[sympy.geometry.line.Segment2D](#) (page 602), [sympy.geometry.line.Segment3D](#) (page 607), [sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Line](#) (page 589)

Notes

If 2D or 3D points are used to define *Segment*, it will be automatically subclassed to *Segment2D* or *Segment3D*.

Examples

```
>>> from sympy import Point
>>> from sympy.geometry import Segment
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment2D(Point2D(1, 0), Point2D(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1))
>>> s.points
(Point2D(4, 3), Point2D(1, 1))
>>> s.slope
```

(continues on next page)

(continued from previous page)

```

2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
>>> Segment((1, 0, 0), (1, 1, 1)) # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment(Point(4, 3, 9), Point(1, 1, 7)); s
Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.points
(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)

```

Attributes

length	(number or sympy expression)
midpoint	(Point)

contains(other)

Is the other GeometryEntity contained within this Segment?

Examples

```

>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s2 = Segment(p2, p1)
>>> s.contains(s2)
True
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 1, 1), Point3D(3, 4, 5)
>>> s = Segment3D(p1, p2)
>>> s2 = Segment3D(p2, p1)
>>> s.contains(s2)
True
>>> s.contains((p1 + p2) / 2)
True

```

distance(other)

Finds the shortest distance between a line segment and a point.

Raises `NotImplementedError` is raised if 'other' is not a Point

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s.distance(Point(10, 15))
sqrt(170)
>>> s.distance((0, 12))
sqrt(73)
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 3), Point3D(1, 1, 4)
>>> s = Segment3D(p1, p2)
>>> s.distance(Point3D(10, 15, 12))
sqrt(341)
>>> s.distance((10, 15, 12))
sqrt(341)
```

equals(other)

Returns True if self and other are the same mathematical entities

length

The length of the line segment.

See also:

[sympy.geometry.point.Point.distance](#) (page 569)

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.length
5
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.length
sqrt(34)
```

midpoint

The midpoint of the line segment.

See also:

[sympy.geometry.point.Point.midpoint](#) (page 571)

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.midpoint
Point2D(2, 3/2)
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
```

(continues on next page)

(continued from previous page)

```
>>> s1.midpoint
Point3D(2, 3/2, 3/2)
```

perpendicular_bisector(p=None)

The perpendicular bisector of this segment.

If no point is specified or the point specified is not on the bisector then the bisector is returned as a Line. Otherwise a Segment is returned that joins the point specified and the intersection of the bisector and the segment.

Parameters **p** : Point

Returns **bisector** : Line or Segment

See also:

[LinearEntity.perpendicular_segment](#) (page 586)

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2, p3 = Point(0, 0), Point(6, 6), Point(5, 1)
>>> s1 = Segment(p1, p2)
>>> s1.perpendicular_bisector()
Line2D(Point2D(3, 3), Point2D(-3, 9))
```

```
>>> s1.perpendicular_bisector(p3)
Segment2D(Point2D(5, 1), Point2D(3, 3))
```

plot_interval(parameter='t')

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

Parameters **parameter** : str, optional

Default value is 't'.

Returns **plot_interval** : list

[parameter, lower_bound, upper_bound]

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> s1 = Segment(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

class `sympy.geometry.line.LinearEntity2D`

A base class for all linear entities (line, ray and segment) in a 2-dimensional Euclidean space.

See also:

[sympy.geometry.entity.GeometryEntity](#) (page 561)

Notes

This is an abstract class and is not meant to be instantiated.

Attributes

p1	
p2	
coefficients	
slope	
points	

bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

perpendicular_line(p)

Create a new Line perpendicular to this linear entity which passes through the point p .

Parameters p : Point

Returns line : Line

See also:

is_perpendicular, perpendicular_segment

Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

slope

The slope of this linear entity, or infinity if vertical.

Returns slope : number or sympy expression

See also:

coefficients

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
```

(continues on next page)

(continued from previous page)

```
>>> l1.slope
5/3
```

```
>>> p3 = Point(0, 4)
>>> l2 = Line(p1, p3)
>>> l2.slope
oo
```

class sympy.geometry.line.Line2D

An infinite line in space 2D.

A line is declared with two distinct points or a point and slope as defined using keyword *slope*.

Parameters **p1** : Point

pt : Point

slope : sympy expression

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point
>>> from sympy.abc import L
>>> from sympy.geometry import Line, Segment
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line2D(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword slope:

```
>>> Line(Point(0, 0), slope=0)
Line2D(Point2D(0, 0), Point2D(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

coefficients

The coefficients (a, b, c) for $ax + by + c = 0$.

See also:

`sympy.geometry.line.Line.equation`

Examples

```
>>> from sympy import Point, Line
>>> from sympy.abc import x, y
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.coefficients
(-3, 5, 0)
```

```
>>> p3 = Point(x, y)
>>> l2 = Line(p1, p3)
>>> l2.coefficients
(-y, x, 0)
```

equation(x='x', y='y')

The equation of the line: $ax + by + c$.

Parameters **x** : str, optional

The name to use for the x-axis, default value is 'x'.

y : str, optional

The name to use for the y-axis, default value is 'y'.

Returns **equation** : sympy expression

See also:

`LinearEntity.coefficients`

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.equation()
-3*x + 4*y + 3
```

class sympy.geometry.line.Ray2D

A Ray is a semi-line in the space with a source point and a direction.

Parameters **p1** : Point

The source of the Ray

p2 : Point or radian value

This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.

See also:

[sympy.geometry.point.Point](#) (page 567), [Line](#) (page 589)

Examples

```
>>> from sympy import Point, pi
>>> from sympy.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray2D(Point2D(2, 3), Point2D(3, 5))
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
>>> r.source
Point2D(2, 3)
>>> r.xdirection
00
>>> r.ydirection
00
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

Attributes

source	
xdirection	
ydirection	

closing_angle(r2)

Return the angle by which r2 must be rotated so it faces the same direction as r1.

Parameters **r1** : Ray2D

r2 : Ray2D

Returns **angle** : angle in radians (ccw angle is positive)

See also:

[LinearEntity.angle_between](#) (page 580)

Examples

```
>>> from sympy import Ray, pi
>>> r1 = Ray((0, 0), (1, 0))
>>> r2 = r1.rotate(-pi/2)
>>> angle = r1.closing_angle(r2); angle
pi/2
>>> r2.rotate(angle).direction.unit == r1.direction.unit
True
>>> r2.closing_angle(r1)
-pi/2
```

xdirection

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

See also:

[ydirection](#) (page 602)

Examples

```
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, -1)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

ydirection

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 602)

Examples

```
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(-1, -1), Point(-1, 0)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

class sympy.geometry.line.Segment2D

A line segment in 2D space.

Parameters **p1** : Point

p2 : Point

See also:

`sympy.geometry.point.Point` (page 567), `Line` (page 589)

Examples

```
>>> from sympy import Point
>>> from sympy.geometry import Segment
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment2D(Point2D(1, 0), Point2D(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1)); s
Segment2D(Point2D(4, 3), Point2D(1, 1))
>>> s.points
(Point2D(4, 3), Point2D(1, 1))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
```

Attributes

length	(number or sympy expression)
midpoint	(Point)

class `sympy.geometry.line.LinearEntity3D`

An base class for all linear entities (line, ray and segment) in a 3-dimensional Euclidean space.

Notes

This is a base class and is not meant to be instantiated.

Attributes

p1	
p2	
direction_ratio	
direction_cosine	
points	

direction_cosine

The normalized direction ratio of a given line in 3D.

See also:

`sympy.geometry.line.Line.equation`

Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_cosine
[sqrt(35)/7, 3*sqrt(35)/35, sqrt(35)/35]
>>> sum(i**2 for i in _)
1
```

direction_ratio

The direction ratio of a given line in 3D.

See also:

`sympy.geometry.line.Line.equation`

Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_ratio
[5, 3, 1]
```

class `sympy.geometry.line.Line3D`

An infinite 3D line in space.

A line is declared with two distinct points or a point and `direction_ratio` as defined using keyword *direction_ratio*.

Parameters `p1` : Point3D

`pt` : Point3D

`direction_ratio` : list

See also:

`sympy.geometry.point.Point3D` (page 575), `sympy.geometry.line.Line` (page 589), `sympy.geometry.line.Line2D` (page 599)

Examples

```
>>> from sympy import Point3D
>>> from sympy.geometry import Line3D, Segment3D
>>> L = Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L
Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L.points
(Point3D(2, 3, 4), Point3D(3, 5, 1))
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

equation(x='x', y='y', z='z', k=None)

Return the equations that define the line in 3D.

Parameters **x** : str, optional

The name to use for the x-axis, default value is 'x'.

y : str, optional

The name to use for the y-axis, default value is 'y'.

z : str, optional

The name to use for the z-axis, default value is 'z'.

Returns **equation** : Tuple of simultaneous equations

Examples

```
>>> from sympy import Point3D, Line3D, solve
>>> from sympy.abc import x, y, z
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 0)
>>> l1 = Line3D(p1, p2)
>>> eq = l1.equation(x, y, z); eq
(-3*x + 4*y + 3, z)
>>> solve(eq.subs(z, 0), (x, y, z))
{x: 4*y/3 + 1}
```

class sympy.geometry.line.Ray3D

A Ray is a semi-line in the space with a source point and a direction.

Parameters **p1** : Point3D

The source of the Ray

p2 : Point or a direction vector

direction_ratio: Determines the direction in which the Ray propagates.

See also:

[sympy.geometry.point.Point3D](#) (page 575), [Line3D](#) (page 604)

Examples

```
>>> from sympy import Point3D
>>> from sympy.geometry import Ray3D
>>> r = Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r
Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
```

(continues on next page)

(continued from previous page)

```
>>> r.points
(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.source
Point3D(2, 3, 4)
>>> r.xdirection
oo
>>> r.ydirection
oo
>>> r.direction_ratio
[1, 2, -4]
```

Attributes

source	
xdirection	
ydirection	
zdirection	

xdirection

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

See also:

[ydirection](#) (page 606)

Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, -1, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

ydirection

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 606)

Examples

```

>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0

```

zdirection

The z direction of the ray.

Positive infinity if the ray points in the positive z direction, negative infinity if the ray points in the negative z direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 606)

Examples

```

>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
>>> r2.zdirection
0

```

class sympy.geometry.line.Segment3D

A line segment in a 3D space.

Parameters **p1** : Point3D

p2 : Point3D

See also:

[sympy.geometry.point.Point3D](#) (page 575), [Line3D](#) (page 604)

Examples

```

>>> from sympy import Point3D
>>> from sympy.geometry import Segment3D
>>> Segment3D((1, 0, 0), (1, 1, 1)) # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7)); s
Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.points
(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)

```

Attributes

length	(number or sympy expression)
midpoint	(Point3D)

Curves

class sympy.geometry.curve.Curve

A curve in space.

A curve is defined by parametric functions for the coordinates, a parameter and the lower and upper bounds for the parameter value.

Parameters **function** : list of functions

limits : 3-tuple

Function parameter and lower and upper bounds.

Raises **ValueError**

When *functions* are specified incorrectly. When *limits* are specified incorrectly.

See also:

[sympy.core.function.Function](#) (page 183), [sympy.polys.polyfuncs.interpolate](#) (page 865)

Examples

```
>>> from sympy import sin, cos, Symbol, interpolate
>>> from sympy.abc import t, a
>>> from sympy.geometry import Curve
>>> C = Curve((sin(t), cos(t)), (t, 0, 2))
>>> C.functions
(sin(t), cos(t))
>>> C.limits
(t, 0, 2)
>>> C.parameter
t
>>> C = Curve((t, interpolate([1, 4, 9, 16], t)), (t, 0, 1)); C
Curve((t, t**2), (t, 0, 1))
>>> C.subs(t, 4)
Point2D(4, 16)
>>> C.arbitrary_point(a)
Point2D(a, a**2)
```

Attributes

functions	
parameter	
limits	

arbitrary_point(parameter='t')

A parameterized point on the curve.

Parameters **parameter** : str or Symbol, optional

Default value is 't'; the Curve's parameter is selected with None or self.parameter otherwise the provided symbol is used.

Returns **arbitrary_point** : Point

Raises **ValueError**

When *parameter* already appears in the functions.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import s
>>> from sympy.geometry import Curve
>>> C = Curve([2*s, s**2], (s, 0, 2))
>>> C.arbitrary_point()
Point2D(2*t, t**2)
>>> C.arbitrary_point(C.parameter)
Point2D(2*s, s**2)
>>> C.arbitrary_point(None)
Point2D(2*s, s**2)
>>> C.arbitrary_point(Symbol('a'))
Point2D(2*a, a**2)
```

free_symbols

Return a set of symbols other than the bound symbols used to parametrically define the Curve.

Examples

```
>>> from sympy.abc import t, a
>>> from sympy.geometry import Curve
>>> Curve((t, t**2), (t, 0, 2)).free_symbols
set()
>>> Curve((t, t**2), (t, a, 2)).free_symbols
{a}
```

functions

The functions specifying the curve.

Returns **functions** : list of parameterized coordinate functions.

See also:

[parameter](#) (page 610)

Examples

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.functions
(t, t**2)
```

length

The curve length.

Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy import cos, sin
>>> from sympy.abc import t
>>> Curve((t, t), (t, 0, 1)).length
sqrt(2)
```

limits

The limits for the curve.

Returns limits : tuple

Contains parameter and lower and upper limits.

See also:

[plot_interval](#) (page 610)

Examples

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve([t, t**3], (t, -2, 2))
>>> C.limits
(t, -2, 2)
```

parameter

The curve function variable.

Returns parameter : SymPy symbol

See also:

[functions](#) (page 609)

Examples

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve([t, t**2], (t, 0, 2))
>>> C.parameter
t
```


plot_interval(parameter='t')

The plot interval for the default geometric plot of the curve.

Parameters **parameter** : str or Symbol, optional

Default value is 't'; otherwise the provided symbol is used.

Returns **plot_interval** : list (plot interval)

[parameter, lower_bound, upper_bound]

See also:

limits (page 610) Returns limits of the parameter interval

Examples

```
>>> from sympy import Curve, sin
>>> from sympy.abc import x, t, s
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval()
[t, 1, 2]
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval(s)
[s, 1, 2]
```

rotate(angle=0, pt=None)

Rotate angle radians counterclockwise about Point pt.

The default pt is the origin, Point(0, 0).

Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy.abc import x
>>> from sympy import pi
>>> Curve((x, x), (x, 0, 1)).rotate(pi/2)
Curve((-x, x), (x, 0, 1))
```

scale(x=1, y=1, pt=None)

Override GeometryEntity.scale since Curve is not made up of Points.

Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy import pi
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).scale(2)
Curve((2*x, x), (x, 0, 1))
```

translate(x=0, y=0)

Translate the Curve by (x, y).

Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy import pi
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).translate(1, 2)
Curve((x + 1, x + 2), (x, 0, 1))
```

Ellipses

class sympy.geometry.ellipse.Ellipse

An elliptical GeometryEntity.

Parameters *center* : Point, optional

Default value is Point(0, 0)

hradius : number or SymPy expression, optional

vradius : number or SymPy expression, optional

eccentricity : number or SymPy expression, optional

Two of *hradius*, *vradius* and *eccentricity* must be supplied to create an Ellipse. The third is derived from the two supplied.

Raises **GeometryError**

When *hradius*, *vradius* and *eccentricity* are incorrectly supplied as parameters.

TypeError

When *center* is not a Point.

See also:

[Circle](#) (page 624)

Notes

Constructed from a center and two radii, the first being the horizontal radius (along the x-axis) and the second being the vertical radius (along the y-axis).

When symbolic value for *hradius* and *vradius* are used, any calculation that refers to the foci or the major or minor axis will assume that the ellipse has its major radius on the x-axis. If this is not true then a manual rotation is necessary.

Examples

```
>>> from sympy import Ellipse, Point, Rational
>>> e1 = Ellipse(Point(0, 0), 5, 1)
>>> e1.hradius, e1.vradius
(5, 1)
>>> e2 = Ellipse(Point(3, 1), hradius=3, eccentricity=Rational(4, 5))
>>> e2
Ellipse(Point2D(3, 1), 3, 9/5)
```

Attributes

center	
hradius	
vradius	
area	
circumference	
eccentricity	
periapsis	
apoapsis	
focus_distance	
foci	

apoapsis

The apoapsis of the ellipse.

The greatest distance between the focus and the contour.

Returns **apoapsis** : number

See also:

periapsis (page 620) Returns shortest distance between foci and contour

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.apoapsis
2*sqrt(2) + 3
```

arbitrary_point(parameter='t')

A parameterized point on the ellipse.

Parameters **parameter** : str, optional

Default value is 't'.

Returns **arbitrary_point** : Point

Raises **ValueError**

When *parameter* already appears in the functions.

See also:

`sympy.geometry.point.Point` (page 567)

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.arbitrary_point()
Point2D(3*cos(t), 2*sin(t))
```

area

The area of the ellipse.

Returns area : number

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.area
3*pi
```

bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

center

The center of the ellipse.

Returns center : number

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.center
Point2D(0, 0)
```

circumference

The circumference of the ellipse.

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.circumference
12*elliptic_e(8/9)
```

eccentricity

The eccentricity of the ellipse.

Returns eccentricity : number

Examples

```
>>> from sympy import Point, Ellipse, sqrt
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, sqrt(2))
>>> e1.eccentricity
sqrt(7)/3
```

encloses_point(p)

Return True if p is enclosed by (is inside of) self.

Parameters **p** : Point

Returns **encloses_point** : True, False or None

See also:

[sympy.geometry.point.Point](#) (page 567)

Notes

Being on the border of self is considered False.

Examples

```
>>> from sympy import Ellipse, S
>>> from sympy.abc import t
>>> e = Ellipse((0, 0), 3, 2)
>>> e.encloses_point((0, 0))
True
>>> e.encloses_point(e.arbitrary_point(t).subs(t, S.Half))
False
>>> e.encloses_point((4, 0))
False
```

equation(x='x', y='y')

The equation of the ellipse.

Parameters **x** : str, optional

Label for the x-axis. Default value is 'x'.

y : str, optional

Label for the y-axis. Default value is 'y'.

Returns **equation** : sympy expression

See also:

[arbitrary_point](#) ([page 613](#)) Returns parameterized point on ellipse

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.equation()
y**2/4 + (x/3 - 1/3)**2 - 1
```

evolute(x='x', y='y')

The equation of evolute of the ellipse.

Parameters **x** : str, optional

Label for the x-axis. Default value is 'x'.

y : str, optional

Label for the y-axis. Default value is 'y'.

Returns **equation** : sympy expression

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.evolute()
2**(2/3)*y**(2/3) + (3*x - 3)**(2/3) - 5**(2/3)
```

foci

The foci of the ellipse.

Raises ValueError

When the major and minor axis cannot be determined.

See also:

[sympy.geometry.point.Point](#) (page 567)

focus_distance (page 616) Returns the distance between focus and center

Notes

The foci can only be calculated if the major/minor axes are known.

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.foci
(Point2D(-2*sqrt(2), 0), Point2D(2*sqrt(2), 0))
```

focus_distance

The focal distance of the ellipse.

The distance between the center and one focus.

Returns **focus_distance** : number

See also:[foci](#) (page 616)**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.focus_distance
2*sqrt(2)
```

hradius

The horizontal radius of the ellipse.

Returns **hradius** : number

See also:[vradius](#) (page 624), [major](#) (page 618), [minor](#) (page 619)**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.hradius
3
```

intersection(o)

The intersection of this ellipse and another geometrical entity *o*.

Parameters **o** : GeometryEntity

Returns **intersection** : list of GeometryEntity objects

See also:[sympy.geometry.entity.GeometryEntity](#) (page 561)**Notes**

Currently supports intersections with Point, Line, Segment, Ray, Circle and Ellipse types.

Examples

```
>>> from sympy import Ellipse, Point, Line, sqrt
>>> e = Ellipse(Point(0, 0), 5, 7)
>>> e.intersection(Point(0, 0))
[]
>>> e.intersection(Point(5, 0))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(0,0), Point(0, 1)))
```

(continues on next page)

(continued from previous page)

```
[Point2D(0, -7), Point2D(0, 7)]
>>> e.intersection(Line(Point(5,0), Point(5, 1)))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(6,0), Point(6, 1)))
[]
>>> e = Ellipse(Point(-1, 0), 4, 3)
>>> e.intersection(Ellipse(Point(1, 0), 4, 3))
[Point2D(0, -3*sqrt(15)/4), Point2D(0, 3*sqrt(15)/4)]
>>> e.intersection(Ellipse(Point(5, 0), 4, 3))
[Point2D(2, -3*sqrt(7)/4), Point2D(2, 3*sqrt(7)/4)]
>>> e.intersection(Ellipse(Point(100500, 0), 4, 3))
[]
>>> e.intersection(Ellipse(Point(0, 0), 3, 4))
[Point2D(3, 0), Point2D(-363/175, -48*sqrt(111)/175), Point2D(-363/175,
↪48*sqrt(111)/175)]
>>> e.intersection(Ellipse(Point(-1, 0), 3, 4))
[Point2D(-17/5, -12/5), Point2D(-17/5, 12/5), Point2D(7/5, -12/5), Point2D(7/
↪5, 12/5)]
```

is_tangent(o)Is *o* tangent to the ellipse?**Parameters** *o* : GeometryEntity

An Ellipse, LinearEntity or Polygon

Returns is_tangent: booleanTrue if *o* is tangent to the ellipse, False otherwise.**Raises** **NotImplementedError**

When the wrong type of argument is supplied.

See also:[tangent_lines](#) (page 623)**Examples**

```
>>> from sympy import Point, Ellipse, Line
>>> p0, p1, p2 = Point(0, 0), Point(3, 0), Point(3, 3)
>>> e1 = Ellipse(p0, 3, 2)
>>> l1 = Line(p1, p2)
>>> e1.is_tangent(l1)
True
```

major

Longer axis of the ellipse (if it can be determined) else hradius.

Returns major : number or expression**See also:**[hradius](#) (page 617), [vradius](#) (page 624), [minor](#) (page 619)

Examples

```
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.major
3
```

```
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).major
a
>>> Ellipse(p1, b, a).major
b
```

```
>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).major
m + 1
```

minor

Shorter axis of the ellipse (if it can be determined) else vradius.

Returns **minor** : number or expression

See also:

[hradius](#) (page 617), [vradius](#) (page 624), [major](#) (page 618)

Examples

```
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.minor
1
```

```
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).minor
b
>>> Ellipse(p1, b, a).minor
a
```

```
>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).minor
m
```

normal_lines(p, prec=None)

Normal lines between p and the ellipse.

Parameters **p** : Point

Returns **normal_lines** : list with 1, 2 or 4 Lines

Examples

```
>>> from sympy import Line, Point, Ellipse
>>> e = Ellipse((0, 0), 2, 3)
>>> c = e.center
>>> e.normal_lines(c + Point(1, 0))
[Line2D(Point2D(0, 0), Point2D(1, 0))]
>>> e.normal_lines(c)
[Line2D(Point2D(0, 0), Point2D(0, 1)), Line2D(Point2D(0, 0), Point2D(1, 0))]
```

Off-axis points require the solution of a quartic equation. This often leads to very large expressions that may be of little practical use. An approximate solution of *prec* digits can be obtained by passing in the desired value:

```
>>> e.normal_lines((3, 3), prec=2)
[Line2D(Point2D(-0.81, -2.7), Point2D(0.19, -1.2)),
Line2D(Point2D(1.5, -2.0), Point2D(2.5, -2.7))]
```

Whereas the above solution has an operation count of 12, the exact solution has an operation count of 2020.

periapsis

The periapsis of the ellipse.

The shortest distance between the focus and the contour.

Returns **periapsis** : number

See also:

apoapsis (page 613) Returns greatest distance between focus and contour

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.periapsis
-2*sqrt(2) + 3
```

plot_interval(parameter='t')

The plot interval for the default geometric plot of the Ellipse.

Parameters **parameter** : str, optional

Default value is 't'.

Returns **plot_interval** : list

[parameter, lower_bound, upper_bound]

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.plot_interval()
[t, -pi, pi]
```

random_point(seed=None)
A random point on the ellipse.

Returns point : Point

See also:

[sympy.geometry.point.Point](#) (page 567)

arbitrary_point (page 613) Returns parameterized point on ellipse

Notes

When creating a random point, one may simply replace the parameter with a random number. When doing so, however, the random number should be made a Rational or else the point may not test as being in the ellipse:

```
>>> from sympy.abc import t
>>> from sympy import Rational
>>> arb = e1.arbitrary_point(t); arb
Point2D(3*cos(t), 2*sin(t))
>>> arb.subs(t, .1) in e1
False
>>> arb.subs(t, Rational(.1)) in e1
True
>>> arb.subs(t, Rational('.1')) in e1
True
```

Examples

```
>>> from sympy import Point, Ellipse, Segment
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.random_point() # gives some random point
Point2D(...)
>>> p1 = e1.random_point(seed=0); p1.n(2)
Point2D(2.1, 1.4)
```

reflect(line)
Override GeometryEntity.reflect since the radius is not a GeometryEntity.

Notes

Until the general ellipse (with no axis parallel to the x-axis) is supported a NotImplemented error is raised and the equation whose zeros define the rotated ellipse is given.

Examples

```
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
>>> from sympy import Ellipse, Line, Point
```

(continues on next page)

(continued from previous page)

```
>>> Ellipse(Point(3, 4), 1, 3).reflect(Line(Point(0, -4), Point(5, 0)))
Traceback (most recent call last):
...
NotImplementedError:
General Ellipse is not supported but the equation of the reflected
Ellipse is given by the zeros of:  $f(x, y) = (9x/41 + 40y/41 + 37/41)**2 + (40x/123 - 3y/41 - 364/123)**2 - 1$ 
```

rotate(angle=0, pt=None)

Rotate angle radians counterclockwise about Point pt.

Note: since the general ellipse is not supported, only rotations that are integer multiples of $\pi/2$ are allowed.

Examples

```
>>> from sympy import Ellipse, pi
>>> Ellipse((1, 0), 2, 1).rotate(pi/2)
Ellipse(Point2D(0, 1), 1, 2)
>>> Ellipse((1, 0), 2, 1).rotate(pi)
Ellipse(Point2D(-1, 0), 2, 1)
```

scale(x=1, y=1, pt=None)

Override GeometryEntity.scale since it is the major and minor axes which must be scaled and they are not GeometryEntities.

Examples

```
>>> from sympy import Ellipse
>>> Ellipse((0, 0), 2, 1).scale(2, 4)
Circle(Point2D(0, 0), 4)
>>> Ellipse((0, 0), 2, 1).scale(2)
Ellipse(Point2D(0, 0), 4, 1)
```

second_moment_of_area(point=None)

Returns the second moment and product moment area of an ellipse.

Parameters point : Point, two-tuple of sympifiable objects, or None(default=None)

point is the point about which second moment of area is to be found. If "point=None" it will be calculated about the axis passing through the centroid of the ellipse.

Returns I_xx, I_yy, I_xy : number or sympy expression

I_xx, I_yy are second moment of area of an ellipse. I_xy is product moment of area of an ellipse.

References

https://en.wikipedia.org/wiki/List_of_second_moments_of_area

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.second_moment_of_area()
(3*pi/4, 27*pi/4, 0)
```

semilatus_rectum

Calculates the semi-latus rectum of the Ellipse.

Semi-latus rectum is defined as one half of the the chord through a focus parallel to the conic section directrix of a conic section.

Returns **semilatus_rectum** : number

See also:

apoapsis (page 613) Returns greatest distance between focus and contour

periapsis (page 620) The shortest distance between the focus and the contour

References

[1] <http://mathworld.wolfram.com/SemilatusRectum.html> [2] https://en.wikipedia.org/wiki/Ellipse#Semi-latus_rectum

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.semilatus_rectum
1/3
```

tangent_lines(p)

Tangent lines between p and the ellipse.

If p is on the ellipse, returns the tangent line through point p . Otherwise, returns the tangent line(s) from p to the ellipse, or None if no tangent line is possible (e.g., p inside ellipse).

Parameters **p** : Point

Returns **tangent_lines** : list with 1 or 2 Lines

Raises **NotImplementedError**

Can only find tangent lines for a point, p , on the ellipse.

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Line](#) (page 589)

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.tangent_lines(Point(3, 0))
[Line2D(Point2D(3, 0), Point2D(3, -12))]
```

vradius

The vertical radius of the ellipse.

Returns **vradius** : number

See also:

[hradius](#) (page 617), [major](#) (page 618), [minor](#) (page 619)

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.vradius
1
```

class `sympy.geometry.ellipse.Circle`

A circle in space.

Constructed simply from a center and a radius, or from three non-collinear points.

Parameters **center** : Point

radius : number or sympy expression

points : sequence of three Points

Raises `GeometryError`

When trying to construct circle from three collinear points. When trying to construct circle from incorrect parameters.

See also:

[Ellipse](#) (page 612), [sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy.geometry import Point, Circle
>>> # a circle constructed from a center and radius
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.hradius, c1.vradius, c1.radius
(5, 5, 5)
```

```
>>> # a circle constructed from three points
>>> c2 = Circle(Point(0, 0), Point(1, 1), Point(1, 0))
>>> c2.hradius, c2.vradius, c2.radius, c2.center
(sqrt(2)/2, sqrt(2)/2, sqrt(2)/2, Point2D(1/2, 1/2))
```

Attributes

radius (synonymous with hradius, vradius, major and minor)	
circumference	
equation	

circumference

The circumference of the circle.

Returns circumference : number or SymPy expression

Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.circumference
12*pi
```

equation(x='x', y='y')

The equation of the circle.

Parameters x : str or Symbol, optional

Default value is 'x'.

y : str or Symbol, optional

Default value is 'y'.

Returns equation : SymPy expression

Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.equation()
x**2 + y**2 - 25
```

intersection(o)

The intersection of this circle with another geometrical entity.

Parameters o : GeometryEntity

Returns intersection : list of GeometryEntities

Examples

```
>>> from sympy import Point, Circle, Line, Ray
>>> p1, p2, p3 = Point(0, 0), Point(5, 5), Point(6, 0)
>>> p4 = Point(5, 0)
>>> c1 = Circle(p1, 5)
>>> c1.intersection(p2)
[]
>>> c1.intersection(p4)
```

(continues on next page)

(continued from previous page)

```
[Point2D(5, 0)]
>>> c1.intersection(Ray(p1, p2))
[Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
>>> c1.intersection(Line(p2, p3))
[]
```

radius

The radius of the circle.

Returns **radius** : number or sympy expression

See also:

[Ellipse.major](#) (page 618), [Ellipse.minor](#) (page 619), [Ellipse.hradius](#) (page 617), [Ellipse.vradius](#) (page 624)

Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.radius
6
```

reflect(line)

Override GeometryEntity.reflect since the radius is not a GeometryEntity.

Examples

```
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
```

scale(x=1, y=1, pt=None)

Override GeometryEntity.scale since the radius is not a GeometryEntity.

Examples

```
>>> from sympy import Circle
>>> Circle((0, 0), 1).scale(2, 2)
Circle(Point2D(0, 0), 2)
>>> Circle((0, 0), 1).scale(2, 4)
Ellipse(Point2D(0, 0), 2, 4)
```

vradius

This Ellipse property is an alias for the Circle's radius.

Whereas hradius, major and minor can use Ellipse's conventions, the vradius does not exist for a circle. It is always a positive value in order that the Circle, like Polygons, will have an area that can be positive or negative as determined by the sign of the hradius.

Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.vradius
6
```

Polygons

class `sympy.geometry.polygon.Polygon`

A two-dimensional polygon.

A simple polygon in space. Can be constructed from a sequence of points or from a center, radius, number of sides and rotation angle.

Parameters `vertices` : sequence of Points

Raises `GeometryError`

If all parameters are not Points.

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Segment](#) (page 594), [Triangle](#) (page 641)

Notes

Polygons are treated as closed paths rather than 2D areas so some calculations can be be negative or positive (e.g., area) based on the orientation of the points.

Any consecutive identical points are reduced to a single point and any points collinear and between two points will be removed unless they are needed to define an explicit intersection (see examples).

A Triangle, Segment or Point will be returned when there are 3 or fewer points provided.

Examples

```
>>> from sympy import Point, Polygon, pi
>>> p1, p2, p3, p4, p5 = [(0, 0), (1, 0), (5, 1), (0, 1), (3, 0)]
>>> Polygon(p1, p2, p3, p4)
Polygon(Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1))
>>> Polygon(p1, p2)
Segment2D(Point2D(0, 0), Point2D(1, 0))
>>> Polygon(p1, p2, p5)
Segment2D(Point2D(0, 0), Point2D(3, 0))
```

The area of a polygon is calculated as positive when vertices are traversed in a ccw direction. When the sides of a polygon cross the area will have positive and negative contributions. The following defines a Z shape where the bottom right connects back to the top left.

```
>>> Polygon((0, 2), (2, 2), (0, 0), (2, 0)).area
0
```

When the keyword n is used to define the number of sides of the Polygon then a RegularPolygon is created and the other arguments are interpreted as center, radius and rotation. The unrotated RegularPolygon will always have a vertex at $\text{Point}(r, 0)$ where r is the radius of the circle that circumscribes the RegularPolygon. Its method *spin* can be used to increment that angle.

```
>>> p = Polygon((0,0), 1, n=3)
>>> p
RegularPolygon(Point2D(0, 0), 1, 3, 0)
>>> p.vertices[0]
Point2D(1, 0)
>>> p.args[0]
Point2D(0, 0)
>>> p.spin(pi/2)
>>> p.vertices[0]
Point2D(0, 1)
```

Attributes

area	
angles	
perimeter	
vertices	
centroid	
sides	

angles

The internal angle at each vertex.

Returns angles : dict

A dictionary where each key is a vertex and each value is the internal angle at that vertex. The vertices are represented as Points.

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.LinearEntity.angle_between](#) (page 580)

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.angles[p1]
pi/2
>>> poly.angles[p2]
acos(-4*sqrt(17)/17)
```

arbitrary_point(parameter='t')

A parameterized point on the polygon.

The parameter, varying from 0 to 1, assigns points to the position on the perimeter that is that fraction of the total perimeter. So the point evaluated at $t=1/2$ would return the point from the first vertex that is 1/2 way around the polygon.

Parameters `parameter` : str, optional

Default value is 't'.

Returns `arbitrary_point` : Point

Raises `ValueError`

When *parameter* already appears in the Polygon's definition.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy import Polygon, S, Symbol
>>> t = Symbol('t', real=True)
>>> tri = Polygon((0, 0), (1, 0), (1, 1))
>>> p = tri.arbitrary_point('t')
>>> perimeter = tri.perimeter
>>> s1, s2 = [s.length for s in tri.sides[:2]]
>>> p.subs(t, (s1 + s2/2)/perimeter)
Point2D(1, 1/2)
```

area

The area of the polygon.

See also:

[sympy.geometry.ellipse.Ellipse.area](#) (page 613)

Notes

The area calculation can be positive or negative based on the orientation of the points. If any side of the polygon crosses any other side, there will be areas having opposite signs.

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.area
3
```

In the Z shaped polygon (with the lower right connecting back to the upper left) the areas cancel out:

```
>>> Z = Polygon((0, 1), (1, 1), (0, 0), (1, 0))
>>> Z.area
0
```

In the M shaped polygon, areas do not cancel because no side crosses any other (though there is a point of contact).

```
>>> M = Polygon((0, 0), (0, 1), (2, 0), (3, 1), (3, 0))
>>> M.area
-3/2
```

bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

centroid

The centroid of the polygon.

Returns centroid : Point

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.util.centroid](#) (page 566)

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.centroid
Point2D(31/18, 11/18)
```

distance(o)

Returns the shortest distance between self and o.

If o is a point, then self does not need to be convex. If o is another polygon self and o must be complex.

Examples

```
>>> from sympy import Point, Polygon, RegularPolygon
>>> p1, p2 = map(Point, [(0, 0), (7, 5)])
>>> poly = Polygon(*RegularPolygon(p1, 1, 3).vertices)
>>> poly.distance(p2)
sqrt(61)
```

encloses_point(p)

Return True if p is enclosed by (is inside of) self.

Parameters **p** : Point

Returns **encloses_point** : True, False or None

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.ellipse.Ellipse.encloses_point](#) (page 615)

Notes

Being on the border of self is considered False.

References

[1] <http://paulbourke.net/geometry/polygonmesh/#insidepoly>

Examples

```
>>> from sympy import Polygon, Point
>>> from sympy.abc import t
>>> p = Polygon((0, 0), (4, 0), (4, 4))
>>> p.encloses_point(Point(2, 1))
True
>>> p.encloses_point(Point(2, 2))
False
>>> p.encloses_point(Point(5, 5))
False
```

intersection(o)

The intersection of polygon and geometry entity.

The intersection may be empty and can contain individual Points and complete Line Segments.

Parameters other: GeometryEntity

Returns intersection : list

The list of Segments and Points

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Segment](#) (page 594)

Examples

```
>>> from sympy import Point, Polygon, Line
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly1 = Polygon(p1, p2, p3, p4)
>>> p5, p6, p7 = map(Point, [(3, 2), (1, -1), (0, 2)])
>>> poly2 = Polygon(p5, p6, p7)
>>> poly1.intersection(poly2)
[Point2D(1/3, 1), Point2D(2/3, 0), Point2D(9/5, 1/5), Point2D(7/3, 1)]
>>> poly1.intersection(Line(p1, p2))
[Segment2D(Point2D(0, 0), Point2D(1, 0))]
>>> poly1.intersection(p1)
[Point2D(0, 0)]
```

is_convex()

Is the polygon convex?

A polygon is convex if all its interior angles are less than 180 degrees and there are no intersections between sides.

Returns is_convex : boolean

True if this polygon is convex, False otherwise.

See also:

[sympy.geometry.util.convex_hull](#) (page 565)

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.is_convex()
True
```

perimeter

The perimeter of the polygon.

Returns **perimeter** : number or Basic instance

See also:

[sympy.geometry.line.Segment.length](#) (page 596)

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.perimeter
sqrt(17) + 7
```

plot_interval(parameter='t')

The plot interval for the default geometric plot of the polygon.

Parameters **parameter** : str, optional

Default value is 't'.

Returns **plot_interval** : list (plot interval)

[parameter, lower_bound, upper_bound]

Examples

```
>>> from sympy import Polygon
>>> p = Polygon((0, 0), (1, 0), (1, 1))
>>> p.plot_interval()
[t, 0, 1]
```

second_moment_of_area(point=None)

Returns the second moment and product moment of area of a two dimensional polygon.

Parameters **point** : Point, two-tuple of sympifiable objects, or None(default=None)

point is the point about which second moment of area is to be found. If "point=None" it will be calculated about the axis passing through the centroid of the polygon.

Returns I_{xx} , I_{yy} , I_{xy} : number or sympy expression

I_{xx} , I_{yy} are second moment of area of a two dimensional polygon.
 I_{xy} is product moment of area of a two dimensional polygon.

References

https://en.wikipedia.org/wiki/Second_moment_of_area

Examples

```
>>> from sympy import Point, Polygon, symbols
>>> a, b = symbols('a, b')
>>> p1, p2, p3, p4, p5 = [(0, 0), (a, 0), (a, b), (0, b), (a/3, b/3)]
>>> rectangle = Polygon(p1, p2, p3, p4)
>>> rectangle.second_moment_of_area()
(a*b**3/12, a**3*b/12, 0)
>>> rectangle.second_moment_of_area(p5)
(a*b**3/9, a**3*b/9, a**2*b**2/36)
```

sides

The directed line segments that form the sides of the polygon.

Returns sides : list of sides

Each side is a directed Segment.

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Segment](#) (page 594)

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.sides
[Segment2D(Point2D(0, 0), Point2D(1, 0)),
 Segment2D(Point2D(1, 0), Point2D(5, 1)),
 Segment2D(Point2D(5, 1), Point2D(0, 1)), Segment2D(Point2D(0, 1), Point2D(0, 0))]
```

vertices

The vertices of the polygon.

Returns vertices : list of Points

See also:

[sympy.geometry.point.Point](#) (page 567)

Notes

When iterating over the vertices, it is more efficient to index self rather than to request the vertices and index them. Only use the vertices when you want to process all of them at once. This is even more important with RegularPolygons that calculate each vertex.

Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.vertices
[Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1)]
>>> poly.vertices[0]
Point2D(0, 0)
```

class sympy.geometry.polygon.RegularPolygon

A regular polygon.

Such a polygon has all internal angles equal and all sides the same length.

Parameters *center* : Point

radius : number or Basic instance

The distance from the center to a vertex

n : int

The number of sides

Raises GeometryError

If the *center* is not a Point, or the *radius* is not a number or Basic instance, or the number of sides, *n*, is less than three.

See also:

[sympy.geometry.point.Point](#) (page 567), [Polygon](#) (page 627)

Notes

A RegularPolygon can be instantiated with Polygon with the kwarg *n*.

Regular polygons are instantiated with a center, radius, number of sides and a rotation angle. Whereas the arguments of a Polygon are vertices, the vertices of the RegularPolygon must be obtained with the vertices method.

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r
RegularPolygon(Point2D(0, 0), 5, 3, 0)
>>> r.vertices[0]
Point2D(5, 0)
```


Attributes

vertices	
center	
radius	
rotation	
apothem	
interior_angle	
exterior_angle	
circumcircle	
incircle	
angles	

angles

Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

Examples

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.angles
{Point2D(-5/2, -5*sqrt(3)/2): pi/3,
 Point2D(-5/2, 5*sqrt(3)/2): pi/3,
 Point2D(5, 0): pi/3}
```

apothem

The inradius of the RegularPolygon.

The apothem/inradius is the radius of the inscribed circle.

Returns apothem : number or instance of Basic

See also:

[sympy.geometry.line.Segment.length](#) (page 596), [sympy.geometry.ellipse.Circle.radius](#) (page 626)

Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.apothem
sqrt(2)*r/2
```

area

Returns the area.

Examples

```
>>> from sympy.geometry import RegularPolygon
>>> square = RegularPolygon((0, 0), 1, 4)
>>> square.area
2
>>> _ == square.length**2
True
```

args

Returns the center point, the radius, the number of sides, and the orientation angle.

Examples

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.args
(Point2D(0, 0), 5, 3, 0)
```

center

The center of the RegularPolygon

This is also the center of the circumscribing circle.

Returns center : Point

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.ellipse.Ellipse.center](#) (page 614)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

centroid

The center of the RegularPolygon

This is also the center of the circumscribing circle.

Returns center : Point

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.ellipse.Ellipse.center](#) (page 614)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

circumcenter

Alias for center.

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.circumcenter
Point2D(0, 0)
```

circumcircle

The circumcircle of the RegularPolygon.

Returns circumcircle : Circle

See also:

[circumcenter](#) (page 637), [sympy.geometry.ellipse.Circle](#) (page 624)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.circumcircle
Circle(Point2D(0, 0), 4)
```

circumradius

Alias for radius.

Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.circumradius
r
```

encloses_point(p)

Return True if p is enclosed by (is inside of) self.

Parameters **p** : Point

Returns encloses_point : True, False or None

See also:

[sympy.geometry.ellipse.Ellipse.encloses_point](#) (page 615)

Notes

Being on the border of self is considered False.

The general Polygon.encloses_point method is called only if a point is not within or beyond the incircle or circumcircle, respectively.

Examples

```
>>> from sympy import RegularPolygon, S, Point, Symbol
>>> p = RegularPolygon((0, 0), 3, 4)
>>> p.encloses_point(Point(0, 0))
True
>>> r, R = p.inradius, p.circumradius
>>> p.encloses_point(Point((r + R)/2, 0))
True
>>> p.encloses_point(Point(R/2, R/2 + (R - r)/10))
False
>>> t = Symbol('t', real=True)
>>> p.encloses_point(p.arbitrary_point().subs(t, S.Half))
False
>>> p.encloses_point(Point(5, 5))
False
```

exterior_angle

Measure of the exterior angles.

Returns exterior_angle : number

See also:

[sympy.geometry.line.LinearEntity.angle_between](#) (page 580)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.exterior_angle
pi/4
```

incircle

The incircle of the RegularPolygon.

Returns incircle : Circle

See also:

[inradius](#) (page 638), [sympy.geometry.ellipse.Circle](#) (page 624)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 7)
>>> rp.incircle
Circle(Point2D(0, 0), 4*cos(pi/7))
```

inradius

Alias for apothem.

Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.inradius
sqrt(2)*r/2
```

interior_angle

Measure of the interior angles.

Returns interior_angle : number

See also:

[sympy.geometry.line.LinearEntity.angle_between](#) (page 580)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.interior_angle
3*pi/4
```

length

Returns the length of the sides.

The half-length of the side and the apothem form two legs of a right triangle whose hypotenuse is the radius of the regular polygon.

Examples

```
>>> from sympy.geometry import RegularPolygon
>>> from sympy import sqrt
>>> s = square_in_unit_circle = RegularPolygon((0, 0), 1, 4)
>>> s.length
sqrt(2)
>>> sqrt((_/2)**2 + s.apothem**2) == s.radius
True
```

radius

Radius of the RegularPolygon

This is also the radius of the circumscribing circle.

Returns radius : number or instance of Basic

See also:

[sympy.geometry.line.Segment.length](#) (page 596), [sympy.geometry.ellipse.Circle.radius](#) (page 626)

Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.radius
r
```

reflect(line)

Override GeometryEntity.reflect since this is not made of only points.

Examples

```
>>> from sympy import RegularPolygon, Line
```

```
>>> RegularPolygon((0, 0), 1, 4).reflect(Line((0, 1), slope=-2))
RegularPolygon(Point2D(4/5, 2/5), -1, 4, atan(4/3))
```

rotate(angle, pt=None)

Override GeometryEntity.rotate to first rotate the RegularPolygon about its center.

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = RegularPolygon(Point(1, 0), 1, 3)
>>> t.vertices[0] # vertex on x-axis
Point2D(2, 0)
>>> t.rotate(pi/2).vertices[0] # vertex on y axis now
Point2D(0, 2)
```

See also:

[rotation](#) (page 640)

spin (page 641) Rotates a RegularPolygon in place

rotation

CCW angle by which the RegularPolygon is rotated

Returns rotation : number or instance of Basic

Examples

```
>>> from sympy import pi
>>> from sympy.abc import a
>>> from sympy.geometry import RegularPolygon, Point
>>> RegularPolygon(Point(0, 0), 3, 4, pi/4).rotation
pi/4
```

Numerical rotation angles are made canonical:

```
>>> RegularPolygon(Point(0, 0), 3, 4, a).rotation
a
>>> RegularPolygon(Point(0, 0), 3, 4, pi).rotation
0
```

scale(x=1, y=1, pt=None)

Override GeometryEntity.scale since it is the radius that must be scaled (if x == y) or else a new Polygon must be returned.

```
>>> from sympy import RegularPolygon
```

Symmetric scaling returns a RegularPolygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 2)
RegularPolygon(Point2D(0, 0), 2, 4, 0)
```

Asymmetric scaling returns a kite as a Polygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 1)
Polygon(Point2D(2, 0), Point2D(0, 1), Point2D(-2, 0), Point2D(0, -1))
```

spin(angle)

Increment in place the virtual Polygon's rotation by ccw angle.

See also: rotate method which moves the center.

```
>>> from sympy import Polygon, Point, pi
>>> r = Polygon(Point(0,0), 1, n=3)
>>> r.vertices[0]
Point2D(1, 0)
>>> r.spin(pi/6)
>>> r.vertices[0]
Point2D(sqrt(3)/2, 1/2)
```

See also:

[rotation](#) (page 640)

rotate (page 640) Creates a copy of the RegularPolygon rotated about a Point

vertices

The vertices of the RegularPolygon.

Returns vertices : list

Each vertex is a Point.

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.vertices
[Point2D(5, 0), Point2D(0, 5), Point2D(-5, 0), Point2D(0, -5)]
```

class sympy.geometry.polygon.Triangle

A polygon with three vertices and three sides.

Parameters points : sequence of Points

keyword: asa, sas, or sss to specify sides/angles of the triangle

Raises GeometryError

If the number of vertices is not equal to three, or one of the vertices is not a Point, or a valid keyword is not given.

See also:

[sympy.geometry.point.Point](#) (page 567), [Polygon](#) (page 627)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
Triangle(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

Keywords `sss`, `sas`, or `asa` can be used to give the desired side lengths (in order) and interior angles (in degrees) that define the triangle:

```
>>> Triangle(sss=(3, 4, 5))
Triangle(Point2D(0, 0), Point2D(3, 0), Point2D(3, 4))
>>> Triangle(asa=(30, 1, 30))
Triangle(Point2D(0, 0), Point2D(1, 0), Point2D(1/2, sqrt(3)/6))
>>> Triangle(sas=(1, 45, 2))
Triangle(Point2D(0, 0), Point2D(2, 0), Point2D(sqrt(2)/2, sqrt(2)/2))
```

Attributes

vertices	
altitudes	
orthocenter	
circumcenter	
circumradius	
circumcircle	
inradius	
incircle	
exradii	
medians	
medial	
nine_point_circle	

altitudes

The altitudes of the triangle.

An altitude of a triangle is a segment through a vertex, perpendicular to the opposite side, with length being the height of the vertex measured from the line containing the side.

Returns altitudes : dict

The dictionary consists of keys which are vertices and values which are Segments.

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Segment.length](#) (page 596)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.altitudes[p1]
Segment2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

bisectors()

The angle bisectors of the triangle.

An angle bisector of a triangle is a straight line through a vertex which cuts the corresponding angle in half.

Returns bisectors : dict

Each key is a vertex (Point) and each value is the corresponding bisector (Segment).

See also:

[sympy.geometry.point.Point](#) (page 567), [sympy.geometry.line.Segment](#) (page 594)

Examples

```
>>> from sympy.geometry import Point, Triangle, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> from sympy import sqrt
>>> t.bisectors()[p2] == Segment(Point(1, 0), Point(0, sqrt(2) - 1))
True
```

circumcenter

The circumcenter of the triangle

The circumcenter is the center of the circumcircle.

Returns circumcenter : Point

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcenter
Point2D(1/2, 1/2)
```

circumcircle

The circle which passes through the three vertices of the triangle.

Returns circumcircle : Circle

See also:

[sympy.geometry.ellipse.Circle](#) (page 624)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcircle
Circle(Point2D(1/2, 1/2), sqrt(2)/2)
```

circumradius

The radius of the circumcircle of the triangle.

Returns circumradius : number of Basic instance

See also:

[sympy.geometry.ellipse.Circle.radius](#) (page 626)

Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import Point, Triangle
>>> a = Symbol('a')
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, a)
>>> t = Triangle(p1, p2, p3)
>>> t.circumradius
sqrt(a**2/4 + 1/4)
```

eulerline

The Euler line of the triangle.

The line which passes through circumcenter, centroid and orthocenter.

Returns eulerline : Line (or Point for equilateral triangles in which case all centers coincide)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.eulerline
Line2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

exradii

The radius of excircles of a triangle.

An excircle of the triangle is a circle lying outside the triangle, tangent to one of its sides and tangent to the extensions of the other two.

Returns exradii : dict

See also:

`sympy.geometry.polygon.Triangle.inradius` (page 646)

References

[1] <http://mathworld.wolfram.com/Exradius.html> [2] <http://mathworld.wolfram.com/Excircles.html>

Examples

The exradius touches the side of the triangle to which it is keyed, e.g. the exradius touching side 2 is:

```
>>> from sympy.geometry import Point, Triangle, Segment2D, Point2D
>>> p1, p2, p3 = Point(0, 0), Point(6, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.exradii[t.sides[2]]
-2 + sqrt(10)
```

incenter

The center of the incircle.

The incircle is the circle which lies inside the triangle and touches all three sides.

Returns incenter : Point

See also:

`incircle` (page 645), `sympy.geometry.point.Point` (page 567)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.incenter
Point2D(-sqrt(2)/2 + 1, -sqrt(2)/2 + 1)
```

incircle

The incircle of the triangle.

The incircle is the circle which lies inside the triangle and touches all three sides.

Returns incircle : Circle

See also:

`sympy.geometry.ellipse.Circle` (page 624)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(2, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.incircle
Circle(Point2D(-sqrt(2) + 2, -sqrt(2) + 2), -sqrt(2) + 2)
```

inradius

The radius of the incircle.

Returns inradius : number of Basic instance

See also:

[incircle](#) (page 645), [sympy.geometry.ellipse.Circle.radius](#) (page 626)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(4, 0), Point(0, 3)
>>> t = Triangle(p1, p2, p3)
>>> t.inradius
1
```

is_equilateral()

Are all the sides the same length?

Returns is_equilateral : boolean

See also:

[sympy.geometry.entity.GeometryEntity.is_similar](#) (page 562), [RegularPolygon](#) (page 634), [is_isosceles](#) (page 646), [is_right](#) (page 647), [is_scalene](#) (page 647)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_equilateral()
False
```

```
>>> from sympy import sqrt
>>> t2 = Triangle(Point(0, 0), Point(10, 0), Point(5, 5*sqrt(3)))
>>> t2.is_equilateral()
True
```

is_isosceles()

Are two or more of the sides the same length?

Returns is_isosceles : boolean

See also:

[is_equilateral](#) (page 646), [is_right](#) (page 647), [is_scalene](#) (page 647)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(2, 4))
>>> t1.is_isosceles()
True
```

is_right()

Is the triangle right-angled.

Returns is_right : boolean

See also:

[sympy.geometry.line.LinearEntity.is_perpendicular](#) (page 584),
[is_equilateral](#) (page 646), [is_isosceles](#) (page 646), [is_scalene](#) (page 647)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_right()
True
```

is_scalene()

Are all the sides of the triangle of different lengths?

Returns is_scalene : boolean

See also:

[is_equilateral](#) (page 646), [is_isosceles](#) (page 646), [is_right](#) (page 647)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(1, 4))
>>> t1.is_scalene()
True
```

is_similar(t2)

Is another triangle similar to this one.

Two triangles are similar if one can be uniformly scaled to the other.

Parameters other: **Triangle**

Returns is_similar : boolean

See also:

[sympy.geometry.entity.GeometryEntity.is_similar](#) (page 562)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -3))
>>> t1.is_similar(t2)
True
```

```
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -4))
>>> t1.is_similar(t2)
False
```

medial

The medial triangle of the triangle.

The triangle which is formed from the midpoints of the three sides.

Returns medial : Triangle

See also:

[sympy.geometry.line.Segment.midpoint](#) (page 596)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medial
Triangle(Point2D(1/2, 0), Point2D(1/2, 1/2), Point2D(0, 1/2))
```

medians

The medians of the triangle.

A median of a triangle is a straight line through a vertex and the midpoint of the opposite side, and divides the triangle into two equal areas.

Returns medians : dict

Each key is a vertex (Point) and each value is the median (Segment) at that point.

See also:

[sympy.geometry.point.Point.midpoint](#) (page 571), [sympy.geometry.line.Segment.midpoint](#) (page 596)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medians[p1]
Segment2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

nine_point_circle

The nine-point circle of the triangle.

Nine-point circle is the circumcircle of the medial triangle, which passes through the feet of altitudes and the middle points of segments connecting the vertices and the orthocenter.

Returns `nine_point_circle` : Circle

See also:

[sympy.geometry.line.Segment.midpoint](#) (page 596), [sympy.geometry.polygon.Triangle.medial](#) (page 648), [sympy.geometry.polygon.Triangle.orthocenter](#) (page 649)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.nine_point_circle
Circle(Point2D(1/4, 1/4), sqrt(2)/4)
```

orthocenter

The orthocenter of the triangle.

The orthocenter is the intersection of the altitudes of a triangle. It may lie inside, outside or on the triangle.

Returns `orthocenter` : Point

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.orthocenter
Point2D(0, 0)
```

vertices

The triangle's vertices

Returns `vertices` : tuple

Each element in the tuple is a Point

See also:

[sympy.geometry.point.Point](#) (page 567)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t.vertices
(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

Plane

`class sympy.geometry.plane.Plane`

A plane is a flat, two-dimensional surface. A plane is the two-dimensional analogue of a point (zero-dimensions), a line (one-dimension) and a solid (three-dimensions). A plane can generally be constructed by two types of inputs. They are three non-collinear points and a point and the plane's normal vector.

Examples

```
>>> from sympy import Plane, Point3D
>>> from sympy.abc import x
>>> Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane((1, 1, 1), (2, 3, 4), (2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane(Point3D(1, 1, 1), normal_vector=(1,4,7))
Plane(Point3D(1, 1, 1), (1, 4, 7))
```

Attributes

p1	
normal_vector	

`angle_between(o)`

Angle between the plane and other geometric entity.

Parameters `LinearEntity3D, Plane`.

Returns `angle` : angle in radians

Notes

This method accepts only 3D entities as it's parameter, but if you want to calculate the angle between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the angle.

Examples

```
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 2), normal_vector=(1, 2, 3))
>>> b = Line3D(Point3D(1, 3, 4), Point3D(2, 2, 2))
>>> a.angle_between(b)
-asin(sqrt(21)/6)
```

`arbitrary_point(u=None, v=None)`

Returns an arbitrary point on the Plane. If given two parameters, the point ranges over the entire plane. If given 1 or no parameters, returns a point with one parameter which, when varying from 0 to 2π , moves the point in a circle of radius 1 about p1 of the Plane.

Returns Point3D

Examples

```
>>> from sympy.geometry import Plane, Ray
>>> from sympy.abc import u, v, t, r
>>> p = Plane((1, 1, 1), normal_vector=(1, 0, 0))
>>> p.arbitrary_point(u, v)
Point3D(1, u + 1, v + 1)
>>> p.arbitrary_point(t)
Point3D(1, cos(t) + 1, sin(t) + 1)
```

While arbitrary values of u and v can move the point anywhere in the plane, the single-parameter point can be used to construct a ray whose arbitrary point can be located at angle t and radius r from $p.p1$:

```
>>> Ray(p.p1, _).arbitrary_point(r)
Point3D(1, r*cos(t) + 1, r*sin(t) + 1)
```

static are_concurrent(*planes)

Is a sequence of Planes concurrent?

Two or more Planes are concurrent if their intersections are a common line.

Parameters planes: list

Returns Boolean

Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(5, 0, 0), normal_vector=(1, -1, 1))
>>> b = Plane(Point3D(0, -2, 0), normal_vector=(3, 1, 1))
>>> c = Plane(Point3D(0, -1, 0), normal_vector=(5, -1, 9))
>>> Plane.are_concurrent(a, b)
True
>>> Plane.are_concurrent(a, b, c)
False
```

distance(o)

Distance between the plane and another geometric entity.

Parameters Point3D, LinearEntity3D, Plane.

Returns distance

Notes

This method accepts only 3D entities as it's parameter, but if you want to calculate the distance between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the distance.

Examples

```
>>> from sympy import Point, Point3D, Line, Line3D, Plane
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.distance(b)
sqrt(3)
>>> c = Line3D(Point3D(2, 3, 1), Point3D(1, 2, 2))
>>> a.distance(c)
0
```

equals(o)

Returns True if self and o are the same mathematical entities.

Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Plane(Point3D(1, 2, 3), normal_vector=(2, 2, 2))
>>> c = Plane(Point3D(1, 2, 3), normal_vector=(-1, 4, 6))
>>> a.equals(a)
True
>>> a.equals(b)
True
>>> a.equals(c)
False
```

equation(x=None, y=None, z=None)

The equation of the Plane.

Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 2), Point3D(2, 4, 7), Point3D(3, 5, 1))
>>> a.equation()
-23*x + 11*y - 2*z + 16
>>> a = Plane(Point3D(1, 4, 2), normal_vector=(6, 6, 6))
>>> a.equation()
6*x + 6*y + 6*z - 42
```

intersection(o)

The intersection with other geometrical entity.

Parameters Point, Point3D, LinearEntity, LinearEntity3D, Plane

Returns List

Examples

```
>>> from sympy import Point, Point3D, Line, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
```

(continues on next page)

(continued from previous page)

```

>>> a.intersection(b)
[Point3D(1, 2, 3)]
>>> c = Line3D(Point3D(1, 4, 7), Point3D(2, 2, 2))
>>> a.intersection(c)
[Point3D(2, 2, 2)]
>>> d = Plane(Point3D(6, 0, 0), normal_vector=(2, -5, 3))
>>> e = Plane(Point3D(2, 0, 0), normal_vector=(3, 4, -3))
>>> d.intersection(e)
[Line3D(Point3D(78/23, -24/23, 0), Point3D(147/23, 321/23, 23))]

```

is_coplanar(o)

Returns True if o is coplanar with self, else False.

Examples

```

>>> from sympy import Plane, Point3D
>>> o = (0, 0, 0)
>>> p = Plane(o, (1, 1, 1))
>>> p2 = Plane(o, (2, 2, 2))
>>> p == p2
False
>>> p.is_coplanar(p2)
True

```

is_parallel(l)

Is the given geometric entity parallel to the plane?

Parameters LinearEntity3D or Plane

Returns Boolean

Examples

```

>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(3,1,3), normal_vector=(4, 8, 12))
>>> a.is_parallel(b)
True

```

is_perpendicular(l)

is the given geometric entity perpendicular to the given plane?

Parameters LinearEntity3D or Plane

Returns Boolean

Examples

```

>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(2, 2, 2), normal_vector=(-1, 2, -1))
>>> a.is_perpendicular(b)
True

```

normal_vector

Normal vector of the given plane.

Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.normal_vector
(-1, 2, -1)
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 4, 7))
>>> a.normal_vector
(1, 4, 7)
```

p1

The only defining point of the plane. Others can be obtained from the `arbitrary_point` method.

See also:

[sympy.geometry.point.Point3D](#) (page 575)

Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.p1
Point3D(1, 1, 1)
```

parallel_plane(pt)

Plane parallel to the given plane and passing through the point `pt`.

Parameters `pt`: **Point3D**

Returns **Plane**

Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 4, 6), normal_vector=(2, 4, 6))
>>> a.parallel_plane(Point3D(2, 3, 5))
Plane(Point3D(2, 3, 5), (2, 4, 6))
```

parameter_value(other, u, v=None)

Return the parameter(s) corresponding to the given point.

Examples

```
>>> from sympy import Plane, Point, pi
>>> from sympy.abc import t, u, v
>>> p = Plane((2, 0, 0), (0, 0, 1), (0, 1, 0))
```

By default, the parameter value returned defines a point that is a distance of 1 from the Plane's `p1` value and in line with the given point:

```
>>> on_circle = p.arbitrary_point(t).subs(t, pi/4)
>>> on_circle.distance(p.p1)
1
>>> p.parameter_value(on_circle, t)
{t: pi/4}
```

Moving the point twice as far from `p1` does not change the parameter value:

```
>>> off_circle = p.p1 + (on_circle - p.p1)*2
>>> off_circle.distance(p.p1)
2
>>> p.parameter_value(off_circle, t)
{t: pi/4}
```

If the 2-value parameter is desired, supply the two parameter symbols and a replacement dictionary will be returned:

```
>>> p.parameter_value(on_circle, u, v)
{u: sqrt(10)/10, v: sqrt(10)/30}
>>> p.parameter_value(off_circle, u, v)
{u: sqrt(10)/5, v: sqrt(10)/15}
```

perpendicular_line(pt)

A line perpendicular to the given plane.

Parameters `pt`: **Point3D**

Returns **Line3D**

Examples

```
>>> from sympy import Plane, Point3D, Line3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> a.perpendicular_line(Point3D(9, 8, 7))
Line3D(Point3D(9, 8, 7), Point3D(11, 12, 13))
```

perpendicular_plane(*pts)

Return a perpendicular passing through the given points. If the direction ratio between the points is the same as the Plane's normal vector then, to select from the infinite number of possible planes, a third point will be chosen on the z-axis (or the y-axis if the normal vector is already parallel to the z-axis). If less than two points are given they will be supplied as follows: if no point is given then `pt1` will be `self.p1`; if a second point is not given it will be a point through `pt1` on a line parallel to the z-axis (if the normal is not already the z-axis, otherwise on the line parallel to the y-axis).

Parameters `pts`: **0, 1 or 2 Point3D**

Returns **Plane**

Examples

```
>>> from sympy import Plane, Point3D, Line3D
>>> a, b = Point3D(0, 0, 0), Point3D(0, 1, 0)
>>> Z = (0, 0, 1)
>>> p = Plane(a, normal_vector=Z)
>>> p.perpendicular_plane(a, b)
Plane(Point3D(0, 0, 0), (1, 0, 0))
```

projection(pt)

Project the given point onto the plane along the plane normal.

Parameters Point or Point3D

Returns Point3D

Examples

```
>>> from sympy import Plane, Point, Point3D
>>> A = Plane(Point3D(1, 1, 2), normal_vector=(1, 1, 1))
```

The projection is along the normal vector direction, not the z axis, so (1, 1) does not project to (1, 1, 2) on the plane A:

```
>>> b = Point3D(1, 1)
>>> A.projection(b)
Point3D(5/3, 5/3, 2/3)
>>> _ in A
True
```

But the point (1, 1, 2) projects to (1, 1) on the XY-plane:

```
>>> XY = Plane((0, 0, 0), (0, 0, 1))
>>> XY.projection((1, 1, 2))
Point3D(1, 1, 0)
```

projection_line(line)

Project the given line onto the plane through the normal plane containing the line.

Parameters LinearEntity or LinearEntity3D

Returns Point3D, Line3D, Ray3D or Segment3D

Notes

For the interaction between 2D and 3D lines(segments, rays), you should convert the line to 3D by using this method. For example for finding the intersection between a 2D and a 3D line, convert the 2D line to a 3D line by projecting it on a required plane and then proceed to find the intersection between those lines.

Examples

```
>>> from sympy import Plane, Line, Line3D, Point, Point3D
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Line(Point3D(1, 1), Point3D(2, 2))
>>> a.projection_line(b)
Line3D(Point3D(4/3, 4/3, 1/3), Point3D(5/3, 5/3, -1/3))
>>> c = Line3D(Point3D(1, 1, 1), Point3D(2, 2, 2))
>>> a.projection_line(c)
Point3D(1, 1, 1)
```

random_point(seed=None)

Returns a random point on the Plane.

Returns Point3D

Examples

```
>>> from sympy import Plane
>>> p = Plane((1, 0, 0), normal_vector=(0, 1, 0))
>>> r = p.random_point(seed=42) # seed value is optional
>>> r.n(3)
Point3D(2.29, 0, -1.35)
```

The random point can be moved to lie on the circle of radius 1 centered on p1:

```
>>> c = p.p1 + (r - p.p1).unit
>>> c.distance(p.p1).equals(1)
True
```

5.11 Holonomic Functions

The `holonomic` (page 657) module is intended to deal with holonomic functions along with various operations on them like addition, multiplication, composition, integration and differentiation. The module also implements various kinds of conversions such as converting holonomic functions to a different form and the other way around.

5.11.1 Contents

About Holonomic Functions

This text aims to explain holonomic functions. We assume you have a basic idea of Differential equations and Abstract algebra.

Definition

Holonomic function is a very general type of special function that includes lots of simple known functions as its special cases. In fact the more known hypergeometric function and Meijer G-function are also a special case of it.

A function is called holonomic if it's a solution to an ordinary differential equation having polynomial coefficients only. Since the general solution of a differential equation consists of

a family of functions rather than a single function, holonomic functions are usually defined by a set of initial conditions along with the differential equation.

Let K be a field of characteristic 0. For example, K can be $\mathbb{Q}\mathbb{Q}$ or $\mathbb{R}\mathbb{R}$. A function $f(x)$ will be holonomic if there exists polynomials $p_0, p_1, p_2, \dots, p_r \in K[x]$ such that

$$p_0 \cdot f(x) + p_1 \cdot f^{(1)}(x) + p_2 \cdot f^{(2)}(x) + \dots + p_r \cdot f^{(r)}(x) = 0$$

This differential equation can also be written as $L \cdot f(x) = 0$ where

$$L = p_0 + p_1 \cdot D + p_2 \cdot D^2 + \dots + p_r \cdot D^r$$

Here D is the Differential Operator and L is called the annihilator of the function.

A unique holonomic function can be defined from the annihilator and a set of initial conditions. For instance:

$$\begin{aligned} f(x) &= \exp(x) : L = D - 1, f(0) = 1 \\ f(x) &= \sin(x) : L = D^2 + 1, f(0) = 0, f'(0) = 1 \end{aligned}$$

Other fundamental functions such as $\cos(x)$, $\log(x)$, bessel functions etc. are also holonomic.

The family of holonomic functions is closed under addition, multiplication, integration, composition. This means if two functions are given are holonomic, then the function resulting on applying these operation on them will also be holonomic.

References

https://en.wikipedia.org/wiki/Holonomic_function

Representation of holonomic functions in SymPy

Class `DifferentialOperator` is used to represent the annihilator but we create differential operators easily using the function `DifferentialOperators()`. Class `HolonomicFunction` represents a holonomic function.

Let's explain this with an example:

Take $\sin(x)$ for instance, the differential equation satisfied by it is $y^{(2)}(x) + y(x) = 0$. By definition we conclude it is a holonomic function. The general solution of this ODE is $C_1 \cdot \sin(x) + C_2 \cdot \cos(x)$ but to get $\sin(x)$ we need to provide initial conditions i.e. $y(0) = 0, y^{(1)}(0) = 1$.

To represent the same in this module one needs to provide the differential equation in the form of annihilator. Basically a differential operator is an operator on functions that differentiates them. So $D^n \cdot y(x) = y^{(n)}(x)$ where $y^{(n)}(x)$ denotes n times differentiation of $y(x)$ with respect to x .

So the differential equation can also be written as $D^2 \cdot y(x) + y(x) = 0$ or $(D^2 + 1) \cdot y(x) = 0$. The part left of $y(x)$ is the annihilator i.e. $D^2 + 1$.

So this is how one will represent $\sin(x)$ as a Holonomic Function:

```
>>> from sympy.holonomic import DifferentialOperators, HolonomicFunction
>>> from sympy.abc import x
>>> from sympy import ZZ
>>> R, D = DifferentialOperators(ZZ.old_poly_ring(x), 'D')
>>> HolonomicFunction(D**2 + 1, x, 0, [0, 1])
HolonomicFunction((1) + (1)*D**2, x, 0, [0, 1])
```


The polynomial coefficients will be members of the ring $\mathbb{Z}[x]$ in the example. The D operator returned by the function `DifferentialOperators()` can be used to create annihilators just like SymPy expressions. We currently use the older implementations of rings in SymPy for priority mechanism.

Operations on holonomic functions

Addition and Multiplication

Two holonomic functions can be added or multiplied with the result also a holonomic functions.

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
```

p and q here are holonomic representation of e^x and $\sin(x)$ respectively.

```
>>> p = HolonomicFunction(Dx - 1, x, 0, [1])
>>> q = HolonomicFunction(Dx**2 + 1, x, 0, [0, 1])
```

Holonomic representation of $e^x + \sin(x)$

```
>>> p + q
HolonomicFunction((-1) + (1)*Dx + (-1)*Dx**2 + (1)*Dx**3, x, 0, [1, 2, 1])
```

Holonomic representation of $e^x \cdot \sin(x)$

```
>>> p * q
HolonomicFunction((2) + (-2)*Dx + (1)*Dx**2, x, 0, [0, 1])
```

Integration and Differentiation

`HolonomicFunction.integrate(limits, initcond=False)`
Integrates the given holonomic function.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).integrate((x, 0, x)) # e^x - 1
HolonomicFunction((-1)*Dx + (1)*Dx**2, x, 0, [0, 1])
>>> HolonomicFunction(Dx**2 + 1, x, 0, [1, 0]).integrate((x, 0, x))
HolonomicFunction((1)*Dx + (1)*Dx**3, x, 0, [0, 1, 0])
```

HolonomicFunction.diff(*args)
Differentiation of the given Holonomic function.

See also:

[integrate](#) (page 659)

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).diff().to_expr()
cos(x)
>>> HolonomicFunction(Dx - 2, x, 0, [1]).diff().to_expr()
2*exp(2*x)
```

Composition with polynomials

HolonomicFunction.composition(expr, *args, **kwargs)
Returns function after composition of a holonomic function with an algebraic function. The method can't compute initial conditions for the result by itself, so they can be also be provided.

See also:

[from_hyper](#) (page 664)

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x).composition(x**2, 0, [1]) # e^(x**2)
HolonomicFunction((-2*x) + (1)*Dx, x, 0, [1])
>>> HolonomicFunction(Dx**2 + 1, x).composition(x**2 - 1, 1, [1, 0])
HolonomicFunction((4*x**3) + (-1)*Dx + (x)*Dx**2, x, 1, [1, 0])
```

Convert to holonomic sequence

HolonomicFunction.to_sequence(lb=True)
Finds recurrence relation for the coefficients in the series expansion of the function about x_0 , where x_0 is the point at which the initial condition is stored.

If the point x_0 is ordinary, solution of the form $[(R, n_0)]$ is returned. Where R is the recurrence relation and n_0 is the smallest n for which the recurrence holds true.

If the point x_0 is regular singular, a list of solutions in the format (R, p, n_0) is returned, i.e. $[(R, p, n_0), \dots]$. Each tuple in this vector represents a recurrence relation R associated with

a root of the indicial equation p . Conditions of a different format can also be provided in this case, see the docstring of `HolonomicFunction` class.

If it's not possible to numerically compute a initial condition, it is returned as a symbol C_j , denoting the coefficient of $(x - x_0)^j$ in the power series about x_0 .

See also:

[HolonomicFunction.series](#) (page 661)

References

[R371] (page 1865), [R372] (page 1865)

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_sequence()
[(HolonomicSequence((-1) + (n + 1)Sn, n), u(0) = 1, 0)]
>>> HolonomicFunction((1 + x)*Dx**2 + Dx, x, 0, [0, 1]).to_sequence()
[(HolonomicSequence((n**2) + (n**2 + n)Sn, n), u(0) = 0, u(1) = 1, u(2) = -1/2,
↪ 2)]
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]}).to_sequence()
[(HolonomicSequence((n), n), u(0) = 1, 1/2, 1)]
```

Series expansion

`HolonomicFunction.series`($n=6$, `coefficient=False`, `order=True`, `_recur=None`)

Finds the power series expansion of given holonomic function about x_0 .

A list of series might be returned if x_0 is a regular point with multiple roots of the indicial equation.

See also:

[HolonomicFunction.to_sequence](#) (page 660)

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).series() # e^x
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).series(n=8) # sin(x)
x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)
```

Numerical evaluation

`HolonomicFunction.evalf(points, method='RK4', h=0.05, derivatives=False)`

Finds numerical value of a holonomic function using numerical methods. (RK4 by default). A set of points (real or complex) must be provided which will be the path for the numerical integration.

The path should be given as a list $[x_1, x_2, \dots, x_n]$. The numerical values will be computed at each point in this order $x_1 \rightarrow x_2 \rightarrow x_3 \dots \rightarrow x_n$.

Returns values of the function at $x_1, x_2, \dots, x_n$ in a list.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
```

A straight line on the real axis from (0 to 1)

```
>>> r = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
```

Runge-Kutta 4th order on e^x from 0.1 to 1. Exact solution at 1 is 2.71828182845905

```
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r)
[1.10517083333333, 1.22140257085069, 1.34985849706254, 1.49182424008069,
1.64872063859684, 1.82211796209193, 2.01375162659678, 2.22553956329232,
2.45960141378007, 2.71827974413517]
```

Euler's method for the same

```
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r, method='Euler')
[1.1, 1.21, 1.331, 1.4641, 1.61051, 1.771561, 1.9487171, 2.14358881,
2.357947691, 2.5937424601]
```

One can also observe that the value obtained using Runge-Kutta 4th order is much more accurate than Euler's method.

Convert to a linear combination of hypergeometric functions

`HolonomicFunction.to_hyper(as_list=False, _recur=None)`

Returns a hypergeometric function (or linear combination of them) representing the given holonomic function.

Returns an answer of the form: $a_1 \cdot x^{b_1} \cdot \text{hyper}() + a_2 \cdot x^{b_2} \cdot \text{hyper}() \dots$

This is very useful as one can now use `hyperexpand` to find the symbolic expressions/functions.

See also:

[`from\_hyper`](#) (page 664), [`from\_meijerg`](#) (page 664)

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> # sin(x)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).to_hyper()
x*hyper(), (3/2,), -x**2/4)
>>> # exp(x)
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_hyper()
hyper(), (), x)
```

Convert to a linear combination of Meijer G-functions

HolonomicFunction.to_meijerg()

Returns a linear combination of Meijer G-functions.

See also:

[to_hyper](#) (page 662)

Examples

```
>>> from sympy.holonomic import expr_to_holonomic
>>> from sympy import sin, cos, hyperexpand, log, symbols
>>> x = symbols('x')
>>> hyperexpand(expr_to_holonomic(cos(x) + sin(x)).to_meijerg())
sin(x) + cos(x)
>>> hyperexpand(expr_to_holonomic(log(x)).to_meijerg()).simplify()
log(x)
```

Convert to expressions

HolonomicFunction.to_expr()

Converts a Holonomic Function back to elementary functions.

Examples

```
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(x**2*Dx**2 + x*Dx + (x**2 - 1), x, 0, [0, S(1)/2]).to_expr()
besselj(1, x)
>>> HolonomicFunction((1 + x)*Dx**3 + Dx**2, x, 0, [1, 1, 1]).to_expr()
x*log(x + 1) + log(x + 1) + 1
```

Converting other representations to holonomic

Converting hypergeometric functions

`sympy.holonomic.holonomic.from_hyper(func, x0=0, evalf=False)`
Converts a hypergeometric function to holonomic. `func` is the Hypergeometric Function and `x0` is the point at which initial conditions are required.

Examples

```
>>> from sympy.holonomic.holonomic import from_hyper, DifferentialOperators
>>> from sympy import symbols, hyper, S
>>> x = symbols('x')
>>> from_hyper(hyper([], [S(3)/2], x**2/4))
HolonomicFunction((-x) + (2)*Dx + (x)*Dx**2, x, 1, [sinh(1), -sinh(1) + cosh(1)])
```

Converting Meijer G-functions

`sympy.holonomic.holonomic.from_meijerg(func, x0=0, evalf=False, initcond=True, domain=QQ)`
Converts a Meijer G-function to Holonomic. `func` is the G-Function and `x0` is the point at which initial conditions are required.

Examples

```
>>> from sympy.holonomic.holonomic import from_meijerg, DifferentialOperators
>>> from sympy import symbols, meijerg, S
>>> x = symbols('x')
>>> from_meijerg(meijerg([], []), ([S(1)/2], [0]), x**2/4)
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1/sqrt(pi)])
```

Converting symbolic expressions

`sympy.holonomic.holonomic.expr_to_holonomic(func, x=None, x0=0, y0=None, lenics=None, domain=None, initcond=True)`
Converts a function or an expression to a holonomic function.

Parameters `func`:

The expression to be converted.

`x`:

variable for the function.

`x0`:

point at which initial condition must be computed.

`y0`:

One can optionally provide initial condition if the method isn't able to do it automatically.

lenics:

Number of terms in the initial condition. By default it is equal to the order of the annihilator.

domain:

Ground domain for the polynomials in x appearing as coefficients in the annihilator.

initcond:

Set it false if you don't want the initial conditions to be computed.

See also:

`meijerint._rewritel`, `_convert_poly_rat_alg`, `_create_table`

Examples

```
>>> from sympy.holonomic.holonomic import expr_to_holonomic
>>> from sympy import sin, exp, symbols
>>> x = symbols('x')
>>> expr_to_holonomic(sin(x))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1])
>>> expr_to_holonomic(exp(x))
HolonomicFunction((-1) + (1)*Dx, x, 0, [1])
```

Uses and Current limitations**Integration**

One can perform integrations using holonomic functions by following these steps:

1. Convert the integrand to a holonomic function.
2. Now integrate the holonomic representation of the function.
3. Convert the integral back to expressions.

Examples

```
>>> from sympy.abc import x, a
>>> from sympy import sin
>>> from sympy.holonomic import expr_to_holonomic
>>> expr_to_holonomic(1/(x**2+a), x).integrate(x).to_expr()
atan(x/sqrt(a))/sqrt(a)
>>> expr_to_holonomic(sin(x)/x).integrate(x).to_expr()
Si(x)
```

As you can see in the first example we converted the function to holonomic, integrated the result and then converted back to symbolic expression.

Limitations

1. Converting to expressions is not always possible. The holonomic function should have a hypergeometric series at x_0 .
2. Implementation of converting to holonomic sequence currently doesn't support Frobenius method when the solutions need to have log terms. This happens when at least one pair of the roots of the indicial equation differ by an integer and frobenius method yields linearly dependent series solutions. Since we use this while converting to expressions, sometimes `to_expr()` (page 663) fails.
3. There doesn't seem to be a way for computing indefinite integrals, so `integrate()` (page 659) basically computes $\int_{x_0}^x f(x)dx$ if no limits are given, where x_0 is the point at which initial conditions for the integrand are stored. Sometimes this gives an additional constant in the result. For instance:

```
>>> expr_to_holonomic(sin(x)).integrate(x).to_expr()
-cos(x) + 1
>>> sin(x).integrate(x)
-cos(x)
```

The indefinite integral of $\sin(x)$ is $-\cos(x)$. But the output is $-\cos(x) + 1$ which is $\int_0^x \sin(x)dx$. Although both are considered correct but $-\cos(x)$ is simpler.

5.12 Symbolic Integrals

The `integrals` module in SymPy implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is `integrate()` (page 688)

- `integrate(f, x)` returns the indefinite integral $\int f dx$
- `integrate(f, (x, a, b))` returns the definite integral $\int_a^b f dx$

5.12.1 Examples

SymPy can integrate a vast array of functions. It can integrate polynomial functions:

```
>>> from sympy import *
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> x = Symbol('x')
>>> integrate(x**2 + x + 1, x)
 3      2
x      x
-- + -- + x
3      2
```

Rational functions:

```
>>> integrate(x/(x**2+2*x+1), x)
      1
log(x + 1) + ----
             x + 1
```


Exponential-polynomial functions. These multiplicative combinations of polynomials and the functions `exp`, `cos` and `sin` can be integrated by hand using repeated integration by parts, which is an extremely tedious process. Happily, SymPy will deal with these integrals.

```
>>> integrate(x**2 * exp(x) * cos(x), x)
      2 x      2 x      x      x      x
x *e *sin(x)  x *e *cos(x)  - x*e *sin(x) +  e *sin(x)  -  e *cos(x)
----- + ----- - ----- + ----- - -----
      2      2      2      2      2
```

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

```
>>> integrate(exp(-x**2)*erf(x), x)
      2
\ / pi *erf (x)
-----
      4
```

5.12.2 Integral Transforms

SymPy has special support for definite integrals, and integral transforms.

`sympy.integrals.transforms.mellin_transform(f, x, s, **hints)`

Compute the Mellin transform $F(s)$ of $f(x)$,

$$F(s) = \int_0^{\infty} x^{s-1} f(x) dx.$$

For all “sensible” functions, this converges absolutely in a strip $a < \operatorname{Re}(s) < b$.

The Mellin transform is related via change of variables to the Fourier transform, and also to the (bilateral) Laplace transform.

This function returns $(F, (a, b), \text{cond})$ where F is the Mellin transform of f , (a, b) is the fundamental strip (as above), and cond are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated `MellinTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. If `noconds=False`, then only F will be returned (i.e. not cond , and also not the strip (a, b)).

```
>>> from sympy.integrals.transforms import mellin_transform
>>> from sympy import exp
>>> from sympy.abc import x, s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)
```

See also:

[inverse_mellin_transform](#) (page 667), [laplace_transform](#) (page 668), [fourier_transform](#) (page 669), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673)

`sympy.integrals.transforms.inverse_mellin_transform(F, s, x, strip, **hints)`

Compute the inverse Mellin transform of $F(s)$ over the fundamental strip given by $\text{strip}=(a, b)$.

This can be defined as

$$f(x) = \int_{c-i\infty}^{c+i\infty} x^{-s} F(s) ds,$$

for any c in the fundamental strip. Under certain regularity conditions on F and/or f , this recovers f from its Mellin transform F (and vice versa), for positive real x .

One of a or b may be passed as `None`; a suitable c will be inferred.

If the integral cannot be computed in closed form, this function returns an unevaluated `InverseMellinTransform` object.

Note that this function will assume x to be positive and real, regardless of the sympy assumptions!

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`.

```
>>> from sympy.integrals.transforms import inverse_mellin_transform
>>> from sympy import oo, gamma
>>> from sympy.abc import x, s
>>> inverse_mellin_transform(gamma(s), s, x, (0, oo))
exp(-x)
```

The fundamental strip matters:

```
>>> f = 1/(s**2 - 1)
>>> inverse_mellin_transform(f, s, x, (-oo, -1))
(x/2 - 1/(2*x))*Heaviside(x - 1)
>>> inverse_mellin_transform(f, s, x, (-1, 1))
-x*Heaviside(-x + 1)/2 - Heaviside(x - 1)/(2*x)
>>> inverse_mellin_transform(f, s, x, (1, oo))
(-x/2 + 1/(2*x))*Heaviside(-x + 1)
```

See also:

[mellin_transform](#) (page 667), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673)

`sympy.integrals.transforms.laplace_transform(f, t, s, **hints)`

Compute the Laplace Transform $F(s)$ of $f(t)$,

$$F(s) = \int_0^{\infty} e^{-st} f(t) dt.$$

For all “sensible” functions, this converges absolutely in a half plane $a < \operatorname{Re}(s)$.

This function returns (F, a, cond) where F is the Laplace transform of f , $\operatorname{Re}(s) > a$ is the half-plane of convergence, and cond are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated `LaplaceTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. If `noconds=True`, only F will be returned (i.e. not cond , and also not the plane a).

```
>>> from sympy.integrals import laplace_transform
>>> from sympy.abc import t, s, a
>>> laplace_transform(t**a, t, s)
(s**(-a)*gamma(a + 1)/s, 0, -re(a) < 1)
```

See also:

[inverse_laplace_transform](#) (page 669), [mellin_transform](#) (page 667), [fourier_transform](#) (page 669), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673)

`sympy.integrals.transforms.inverse_laplace_transform(F, s, t, plane=None, **hints)`

Compute the inverse Laplace transform of $F(s)$, defined as

$$f(t) = \int_{c-i\infty}^{c+i\infty} e^{st} F(s) ds,$$

for c so large that $F(s)$ has no singularities in the half-plane $\operatorname{Re}(s) > c - \epsilon$.

The plane can be specified by argument `plane`, but will be inferred if passed as `None`.

Under certain regularity conditions, this recovers $f(t)$ from its Laplace Transform $F(s)$, for non-negative t , and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated `InverseLaplaceTransform` object.

Note that this function will always assume t to be real, regardless of the sympy assumption on t .

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`.

```
>>> from sympy.integrals.transforms import inverse_laplace_transform
>>> from sympy import exp, Symbol
>>> from sympy.abc import s, t
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

See also:

[laplace_transform](#) (page 668), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673)

`sympy.integrals.transforms.fourier_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency Fourier transform of f , defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `FourierTransform` object.

For other Fourier transform conventions, see the function `sympy.integrals.transforms._fourier_transform()`.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import fourier_transform, exp
>>> from sympy.abc import x, k
>>> fourier_transform(exp(-x**2), x, k)
sqrt(pi)*exp(-pi**2*k**2)
>>> fourier_transform(exp(-x**2), x, k, noconds=False)
(sqrt(pi)*exp(-pi**2*k**2), True)
```

See also:

[inverse_fourier_transform](#) (page 670), [sine_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [cosine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

`sympy.integrals.transforms.inverse_fourier_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse Fourier transform of F , defined as

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i x k} dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseFourierTransform` object.

For other Fourier transform conventions, see the function `sympy.integrals.transforms._fourier_transform()`.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import inverse_fourier_transform, exp, sqrt, pi
>>> from sympy.abc import x, k
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x)
exp(-x**2)
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x, noconds=False)
(exp(-x**2), True)
```

See also:

[fourier_transform](#) (page 669), [sine_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [cosine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

`sympy.integrals.transforms.sine_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency sine transform of f , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \sin(2\pi x k) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `SineTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import sine_transform, exp
>>> from sympy.abc import x, k, a
>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*k*exp(-k**2/(4*a))/(4*a**(3/2))
>>> sine_transform(x**(-a), x, k)
2**(-a + 1/2)*k**(a - 1)*gamma(-a/2 + 1)/gamma(a/2 + 1/2)
```

See also:

[fourier_transform](#) (page 669), [inverse_fourier_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [cosine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

`sympy.integrals.transforms.inverse_sine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse sine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \sin(2\pi x k) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseSineTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import inverse_sine_transform, exp, sqrt, gamma, pi
>>> from sympy.abc import x, k, a
>>> inverse_sine_transform(2**((1-2*a)/2)*k**(a - 1)*
...     gamma(-a/2 + 1)/gamma((a+1)/2), k, x)
x**(-a)
>>> inverse_sine_transform(sqrt(2)*k*exp(-k**2/(4*a))/(4*sqrt(a)**3), k, x)
x*exp(-a*x**2)
```

See also:

[fourier_transform](#) (page 669), [inverse_fourier_transform](#) (page 670), [sine_transform](#) (page 670), [cosine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

`sympy.integrals.transforms.cosine_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency cosine transform of f , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \cos(2\pi x k) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `CosineTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import cosine_transform, exp, sqrt, cos
>>> from sympy.abc import x, k, a
>>> cosine_transform(exp(-a*x), x, k)
sqrt(2)*a/(sqrt(pi)*(a**2 + k**2))
>>> cosine_transform(exp(-a*sqrt(x))*cos(a*sqrt(x)), x, k)
a*exp(-a**2/(2*k))/(2*k**(3/2))
```

See also:

[fourier_transform](#) (page 669), [inverse_fourier_transform](#) (page 670), [sine_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667),

[laplace_transform](#) (page 668)

`sympy.integrals.transforms.inverse_cosine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse cosine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \cos(2\pi x k) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseCosineTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import inverse_cosine_transform, exp, sqrt, pi
>>> from sympy.abc import x, k, a
>>> inverse_cosine_transform(sqrt(2)*a/(sqrt(pi)*(a**2 + k**2)), k, x)
exp(-a*x)
>>> inverse_cosine_transform(1/sqrt(k), k, x)
1/sqrt(x)
```

See also:

[fourier_transform](#) (page 669), [inverse_fourier_transform](#) (page 670), [sine_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [cosine_transform](#) (page 671), [hankel_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

`sympy.integrals.transforms.hankel_transform(f, r, k, nu, **hints)`

Compute the Hankel transform of f , defined as

$$F_{\nu}(k) = \int_0^{\infty} f(r) J_{\nu}(kr) r dr.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `HankelTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import gamma, exp, sinh, cosh
>>> from sympy.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

[fourier_transform](#) (page 669), [inverse_fourier_transform](#) (page 670), [sine_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [cosine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [inverse_hankel_transform](#) (page 673), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

`sympy.integrals.transforms.inverse_hankel_transform(F, k, r, nu, **hints)`

Compute the inverse Hankel transform of F defined as

$$f(r) = \int_0^{\infty} F_{\nu}(k) J_{\nu}(kr) k dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseHankelTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()`. Note that for this transform, by default `noconds=True`.

```
>>> from sympy import hankel_transform, inverse_hankel_transform, gamma
>>> from sympy import gamma, exp, sinh, cosh
>>> from sympy.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

[fourier_transform](#) (page 669), [inverse_fourier_transform](#) (page 670), [sine_transform](#) (page 670), [inverse_sine_transform](#) (page 671), [cosine_transform](#) (page 671), [inverse_cosine_transform](#) (page 672), [hankel_transform](#) (page 672), [mellin_transform](#) (page 667), [laplace_transform](#) (page 668)

5.12.3 Internals

There is a general method for calculating antiderivatives of elementary functions, called the Risch algorithm. The Risch algorithm is a decision procedure that can determine whether an elementary solution exists, and in that case calculate it. It can be extended to handle many nonelementary functions in addition to the elementary ones.

SymPy currently uses a simplified version of the Risch algorithm, called the Risch-Norman algorithm. This algorithm is much faster, but may fail to find an antiderivative, although it is

still very powerful. SymPy also uses pattern matching and heuristics to speed up evaluation of some types of integrals, e.g. polynomials.

For non-elementary definite integrals, SymPy uses so-called Meijer G-functions. Details are described here:

Computing Integrals using Meijer G-Functions

This text aims to describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

Overview

The algorithm to compute $\int f(x)dx$ or $\int_0^\infty f(x)dx$ generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function `hyperexpand` (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g. $\int_{-\infty}^\infty$) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.

Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being $0, \pm 1, \infty$). This is not very important when e.g. expanding a single hypergeometric function into named special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set $S = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : S \rightarrow \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives S both a topology and a complex structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps $\text{Exp} : \mathbb{C} \rightarrow S, (x + iy) \mapsto (\exp(x), y)$ and $\text{Log} : S \rightarrow \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point “zero” (0) to S and denote the resulting object S_0 . Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by $\{\text{Exp}(z) : \Re(z) < k\}$, which at least defines a topology. Elements of S_0 shall be called polar numbers. We further define functions $\text{Arg} : S \rightarrow \mathbb{R}, (r, \theta) \mapsto \theta$ and $|\cdot| : S_0 \rightarrow \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from $\mathbb{C}$ to S . We define $\text{Exp}(a)\text{Exp}(b) = \text{Exp}(a+b)$ for $a, b \in \mathbb{C}$, also for $a \in S$ and $b \in \mathbb{C}$ we define $a^b = \text{Exp}(b \text{Log}(a))$. It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g. $(ab)^c = a^c b^c$) which hold in $\mathbb{C}$ only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions $F(z) = \sqrt{1+z}$ and $G(a, b) = \sqrt{a+b}$, where a, b, z are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if $S(z)$ denotes the standard branch of the square root function on $\mathbb{C}$, we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \text{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} \\ -S(p(z)) & : \text{else} \end{cases}$$

(We are omitting $|z| = 1$ here, this does not matter for integration.) Finally we define $G(a, b) = \sqrt{a}F(b/a)$.

Representing Branched Functions on the Argand Plane

Suppose $f : S \rightarrow \mathbb{C}$ is a holomorphic function. We wish to define a function F on (part of) the complex numbers $\mathbb{C}$ that represents f as closely as possible. This process is known as “introducing branch cuts”. In our situation, there is actually a canonical way of doing this (which is adhered to in all of SymPy), as follows: Introduce the “cut complex plane” $C = \mathbb{C} \setminus \mathbb{R}_{\leq 0}$. Define a function $l : C \rightarrow S$ via $re^{i\theta} \mapsto r\text{Exp}(i\theta)$. Here $r > 0$ and $-\pi < \theta \leq \pi$. Then l is holomorphic, and we define $G = f \circ l$. This called “lifting to the principal branch” throughout the SymPy documentation.

Table Lookups and Inverse Mellin Transforms

Suppose we are given an integrand $f(x)$ and are trying to rewrite it as a single G-function. To do this, we first split $f(x)$ into the form $x^s g(x)$ (where $g(x)$ is supposed to be simpler than $f(x)$). This is because multiplicative powers can be absorbed into the G-function later. This splitting is done by `_split_mul(f, x)`. Then we assemble a tuple of functions that occur in f (e.g. if $f(x) = e^x \cos x$, we would assemble the tuple $(\cos, \exp)$). This is done by the function `_mytype(f, x)`. Next we index a lookup table (created using `_create_lookup_table()`) with this tuple. This (hopefully) yields a list of Meijer G-function formulae involving these functions, we then pattern-match all of them. If one fits, we were successful, otherwise not and we have to try something else.

Suppose now we want to rewrite as a product of two G-functions. To do this, we (try to) find all inequivalent ways of splitting $f(x)$ into a product $f_1(x)f_2(x)$. We could try these splittings in any order, but it is often a good idea to minimize (a) the number of powers occurring in $f_i(x)$ and (b) the number of different functions occurring in $f_i(x)$. Thus given e.g. $f(x) = \sin x e^x \sin 2x$ we should try $f_1(x) = \sin x \sin 2x$, $f_2(x) = e^x$ first. All of this is done by the function `_mul_as_two_parts(f)`.

Finally, we can try a recursive Mellin transform technique. Since the Meijer G-function is defined essentially as a certain inverse mellin transform, if we want to write a function $f(x)$ as a G-function, we can compute its mellin transform $F(s)$. If $F(s)$ is in the right form, the G-function expression can be read off. This technique generalises many standard rewritings, e.g. $e^{ax}e^{bx} = e^{(a+b)x}$.

One twist is that some functions don't have mellin transforms, even though they can be written as G-functions. This is true for example for $f(x) = e^x \sin x$ (the function grows too rapidly to have a mellin transform). However if the function is recognised to be analytic, then we can try to compute the mellin-transform of $f(ax)$ for a parameter a , and deduce the G-function

expression by analytic continuation. (Checking for analyticity is easy. Since we can only deal with a certain subset of functions anyway, we only have to filter out those which are not analytic.)

The function `_rewrite_single` does the table lookup and recursive mellin transform. The functions `_rewrite1` and `_rewrite2` respectively use above-mentioned helpers and `_rewrite_single` to rewrite their argument as respectively one or two G-functions.

Applying the Integral Theorems

If the integrand has been recast into G-functions, evaluating the integral is relatively easy. We first do some substitutions to reduce e.g. the exponent of the argument of the G-function to unity (see `_rewrite_saxena_1` and `_rewrite_saxena`, respectively, for one or two G-functions). Next we go through a list of conditions under which the integral theorem applies. It can fail for basically two reasons: either the integral does not exist, or the manipulations in deriving the theorem may not be allowed (for more details, see this [BlogPost](#) (page 1865)).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing e^z satisfies $G(\text{Exp}(2\pi i)z) = G(z)$ for all $z \in \mathcal{S}$. The function `meijerg.get_period()` can be used to discover this, and the function `principal_branch(z, period)` in `functions/elementary/complexes.py` can be used to exploit the information. This is done transparently by the integration code.

The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually stated in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway.):

1.

$$\int_0^\infty G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \eta x \right) dx = \frac{\prod_{j=1}^m \Gamma(b_j + 1) \prod_{j=1}^n \Gamma(-a_j)}{\eta \prod_{j=m+1}^q \Gamma(-b_j) \prod_{j=n+1}^p \Gamma(a_j + 1)}$$

2.

$$\int_0^\infty G_{u,v}^{s,t} \left(\begin{matrix} c_1, \dots, c_u \\ d_1, \dots, d_v \end{matrix} \middle| \sigma x \right) G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \omega x \right) dx = G_{v+p, u+q}^{m+t, n+s} \left(\begin{matrix} a_1, \dots, a_n, -d_1, \dots, -d_v, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, -c_1, \dots, -c_u, b_{m+1}, \dots, b_q \end{matrix} \middle| \frac{\omega}{\sigma} \right)$$

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in SymPy. They are an amalgamation of conditions found in [\[Prudnikov1990\]](#) (page 1869) and [\[Luke1969\]](#) (page 1869); please let us know if you find any errors.

Conditions of Convergence for Integral (1)

We can without loss of generality assume $p \leq q$, since the G-functions of indices m, n, p, q and of indices n, m, q, p can be related easily (see e.g. [\[Luke1969\]](#) (page 1869), section 5.3). We introduce the following notation:

$$\begin{aligned} \xi &= m + n - p \\ \delta &= m + n - \frac{p + q}{2} \end{aligned}$$

$$C_3 : -\Re(b_j) < 1 \text{ for } j = 1, \dots, m$$

$$0 < -\Re(a_j) \text{ for } j = 1, \dots, n$$

$$C_3^* : -\Re(b_j) < 1 \text{ for } j = 1, \dots, q$$

$$0 < -\Re(a_j) \text{ for } j = 1, \dots, p$$

$$C_4 : -\Re(\delta) + \frac{q+1-p}{2} > q-p$$

The convergence conditions will be detailed in several “cases”, numbered one to five. For later use it will be helpful to separate conditions “at infinity” from conditions “at zero”. By conditions “at infinity” we mean conditions that only depend on the behaviour of the integrand for large, positive values of x , whereas by conditions “at zero” we mean conditions that only depend on the behaviour of the integrand on $(0, \epsilon)$ for any $\epsilon > 0$. Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions n “at zero” and “at infinity” both have to be fulfilled, for some n .

These are the conditions “at infinity”:

1.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi \wedge (A \vee B \vee C),$$

where

$$A = 1 \leq n \wedge p < q \wedge 1 \leq m$$

$$B = 1 \leq p \wedge 1 \leq m \wedge q = p+1 \wedge \neg(n=0 \wedge m=p+1)$$

$$C = 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \dots, \left\lceil \frac{\delta}{2} \right\rceil.$$

2.

$$n = 0 \wedge p+1 \leq m \wedge |\arg(\eta)| < \delta\pi$$

3.

$$(p < q \wedge 1 \leq m \wedge \delta > 0 \wedge |\arg(\eta)| = \delta\pi) \vee (p \leq q-2 \wedge \delta = 0 \wedge \arg(\eta) = 0)$$

4.

$$p = q \wedge \delta = 0 \wedge \arg(\eta) = 0 \wedge \eta \neq 0 \wedge \Re\left(\sum_{j=1}^p b_j - a_j\right) < 0$$

5.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi$$

And these are the conditions “at zero”:

1.

$$\eta \neq 0 \wedge C_3$$

2.

$$C_3$$

3.

$$C_3 \wedge C_4$$

4.

$$C_3$$

5.

$$C_3$$

Conditions of Convergence for Integral (2)

We introduce the following notation:

$$b^* = s + t - \frac{u + v}{2}$$

$$c^* = m + n - \frac{p + q}{2}$$

$$\rho = \sum_{j=1}^v d_j - \sum_{j=1}^u c_j + \frac{u - v}{2} + 1$$

$$\mu = \sum_{j=1}^q b_j - \sum_{j=1}^p a_j + \frac{p - q}{2} + 1$$

$$\phi = q - p - \frac{u - v}{2} + 1$$

$$\eta = 1 - (v - u) - \mu - \rho$$

$$\psi = \frac{\pi(q - m - n) + |\arg(\omega)|}{q - p}$$

$$\theta = \frac{\pi(v - s - t) + |\arg(\sigma)|}{v - u}$$

$$\lambda_c = (q - p)|\omega|^{1/(q-p)} \cos \psi + (v - u)|\sigma|^{1/(v-u)} \cos \theta$$

$$\lambda_{s0}(c_1, c_2) = c_1(q - p)|\omega|^{1/(q-p)} \sin \psi + c_2(v - u)|\sigma|^{1/(v-u)} \sin \theta$$

$$\lambda_s = \begin{cases} \lambda_{s0}(-1, -1) \lambda_{s0}(1, 1) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), -1) \lambda_{s0}(\text{sign}(\arg(\omega)), 1) & \text{for } \arg(\omega) \neq 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(-1, \text{sign}(\arg(\sigma))) \lambda_{s0}(1, \text{sign}(\arg(\sigma))) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) \neq 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), \text{sign}(\arg(\sigma))) & \text{otherwise} \end{cases}$$

$$z_0 = \frac{\omega}{\sigma} e^{-i\pi(b^* + c^*)}$$

$$z_1 = \frac{\sigma}{\omega} e^{-i\pi(b^* + c^*)}$$

The following conditions will be helpful:

$$\begin{aligned}
C_1 : & (a_i - b_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, n, j = 1, \dots, m) \\
& \wedge (c_i - d_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, t, j = 1, \dots, s) \\
C_2 : & \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \dots, m, j = 1, \dots, s \\
C_3 : & \Re(a_i + c_j) < 1 \text{ for } i = 1, \dots, n, j = 1, \dots, t \\
C_4 : & (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, t \\
C_5 : & (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, s \\
C_6 : & (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, n \\
C_7 : & (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, m \\
C_8 : & 0 < |\phi| + 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)) \\
C_9 : & 0 < |\phi| - 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)) \\
C_{10} : & |\arg(\sigma)| < \pi b^* \\
C_{11} : & |\arg(\sigma)| = \pi b^* \\
C_{12} : & |\arg(\omega)| < c^* \pi \\
C_{13} : & |\arg(\omega)| = c^* \pi \\
C_{14}^1 : & (z_0 \neq 1 \wedge |\arg(1 - z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu + \rho - u + v) < 1) \\
C_{14}^2 : & (z_1 \neq 1 \wedge |\arg(1 - z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu + \rho - p + q) < 1) \\
C_{14} : & \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2) \\
C_{15} : & \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0) \\
C_{16} : & \int_0^\infty G_{u,v}^{s,t}(\sigma x) dx \text{ converges at infinity} \\
C_{17} : & \int_0^\infty G_{p,q}^{m,n}(\omega x) dx \text{ converges at infinity}
\end{aligned}$$

Note that C_{16} and C_{17} are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1.

$$mnst \neq 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

2.

$$u = v \wedge b^* = 0 \wedge 0 < c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{12}$$

3.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re \mu < 1 \wedge \Re \rho < 1 \wedge \sigma \neq \omega \wedge C_1 \wedge C_2 \wedge C_3$$

4.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

6.

$$q < p \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{10} \wedge C_{13}$$

7.

$$p < q \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{10} \wedge C_{13}$$

8.

$$v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12}$$

9.

$$u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12}$$

10.

$$q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13}$$

11.

$$p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13}$$

12.

$$p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11}$$

13.

$$p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11}$$

14.

$$p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13}$$

15.

$$q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13}$$

16.

$$q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

17.

$$p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

18.

$$t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10}$$

19.

$$s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10}$$

20.

$$n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12}$$

21.

$$m = 0 \wedge 0 < n \wedge 0 < c^* \wedge 0 < \phi \wedge C_1 \wedge C_3 \wedge C_{12}$$

22.

$$st = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

23.

$$mn = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

24.

$$p < m + n \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

25.

$$q < m + n \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

26.

$$p = q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

27.

$$p = q + 1 \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

28.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

30.

$$n = 0 \wedge \phi = 0 \wedge 0 < s + t \wedge 0 < m \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

31.

$$m = 0 \wedge \phi = 0 \wedge v < s + t \wedge 0 < n \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

32.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

36.

$$C_{17} \wedge t = 0 \wedge u < s \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

37.

$$C_{17} \wedge s = 0 \wedge v < t \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

38.

$$C_{16} \wedge n = 0 \wedge p < m \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

39.

$$C_{16} \wedge m = 0 \wedge q < n \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

The Inverse Laplace Transform of a G-function

The inverse laplace transform of a Meijer G-function can be expressed as another G-function. This is a fairly versatile method for computing this transform. However, I could not find the details in the literature, so I work them out here. In [\[Luke1969\]](#) (page 1869), section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument bz , and convergence conditions are also given. However, we need a formula for argument bz^a for rational a .

We are asked to compute

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a) dz,$$

for positive real t . Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

How to compute the integral

We shall work formally for now. Denote by $\Delta(s)$ the product of gamma functions appearing in the definition of G , so that

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s) z^s ds.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} ds dz.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{dz}{2\pi i} ds.$$

The inner integral is easily seen to be $\frac{1}{\Gamma(-as)} \frac{1}{t^{1+as}}$. (Using Cauchy's theorem and Jordan's lemma deform the contour to run from $-\infty$ to $-\infty$, encircling 0 once in the negative sense. For as real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

$$f(t) = \frac{1}{t} \frac{1}{2\pi i} \int_L \Delta(s) \frac{1}{\Gamma(-as)} \left(\frac{b}{t^a}\right)^s ds,$$

which is a so-called Fox H function (of argument $\frac{b}{t^a}$). For rational a , this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for $\Re(as) < -1$. Thus the contour L has to run left of the line $\Re(as) = -1$. Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient b . Hence so is $f(t)$. Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that $a > 0$, then the latter condition is fulfilled if G converges along contours (2) or (3) of [Luke1969] (page 1869), section 5.2, i.e. either $\delta \geq \frac{a}{2}$ or $p \geq 1, p \geq q$.

When the integral exists

Using [Luke1969] (page 1869), section 5.10, for any given meijer G-function we can find a dominant term of the form $z^a e^{bz^c}$ (although this expression might not be the best possible, because of cancellation).

We must thus investigate

$$\lim_{T \rightarrow \infty} \int_{c-iT}^{c+iT} e^{zt} z^a e^{bz^c} dz.$$

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write $z = c + i\tau$. Then $\arg(z) \rightarrow \pm \frac{\pi}{2}$, and so $e^{zt} \sim e^{it\tau}$ (where $\sim$ shall always mean “asymptotically equivalent up to a positive real multiplicative constant”). Also $z^{x+iy} \sim |\tau|^x e^{iy \log |\tau|} e^{\pm x i \frac{\pi}{2}}$.

Set $\omega_{\pm} = b e^{\pm i \Re(c) \frac{\pi}{2}}$. We have three cases:

1. $b = 0$ or $\Re(c) \leq 0$. In this case the integral converges if $\Re(a) \leq -1$.
2. $b \neq 0$, $\Im(c) = 0$, $\Re(c) > 0$. In this case the integral converges if $\Re(\omega_{\pm}) < 0$.
3. $b \neq 0$, $\Im(c) = 0$, $\Re(c) > 0$, $\Re(\omega_{\pm}) \leq 0$, and at least one of $\Re(\omega_{\pm}) = 0$. Here the same condition as in (1) applies.

Implemented G-Function Formulae

An important part of the algorithm is a table expressing various functions as Meijer G-functions. This is essentially a table of Mellin Transforms in disguise. The following automatically generated table shows the formulae currently implemented in SymPy. An entry “generated” means that the corresponding G-function has a variable number of parameters. This table is intended to shrink in future, when the algorithm’s capabilities of deriving new formulae improve. Of course it has to grow whenever a new class of special functions is to be dealt with.

Elementary functions:

$$\begin{aligned} a &= a G_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) + a G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \\ (z^q p + b)^{-a} &= \frac{b^{-a} G_{1,1}^{1,1} \left(\begin{matrix} -a+1 \\ 0 \end{matrix} \middle| \frac{z^q p}{b} \right)}{\Gamma(a)} \\ \frac{-b^a + (z^q p)^a}{z^q p - b} &= \frac{b^{a-1} G_{2,2}^{2,2} \left(\begin{matrix} 0, a \\ 0, a \end{matrix} \middle| \frac{z^q p}{b} \right) \sin(\pi a)}{\pi} \\ (a + \sqrt{z^q p + a^2})^b &= - \frac{a^b b G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ 0 \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{2\sqrt{\pi}} \\ (-a + \sqrt{z^q p + a^2})^b &= \frac{a^b b G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ b \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{2\sqrt{\pi}} \\ \frac{(a + \sqrt{z^q p + a^2})^b}{\sqrt{z^q p + a^2}} &= \frac{a^{b-1} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ 0 \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{\sqrt{\pi}} \\ \frac{(-a + \sqrt{z^q p + a^2})^b}{\sqrt{z^q p + a^2}} &= \frac{a^{b-1} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ b \end{matrix} \middle| \frac{z^q p}{a^2} \right)}{\sqrt{\pi}} \end{aligned}$$

$$\begin{aligned} \left(z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b &= -\frac{a^{\frac{b}{2}} b G_{2,2}^{2,1} \left(\begin{matrix} \frac{b}{2} + 1 & -\frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a}\right)}{2\sqrt{\pi}} \\ \left(-z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b &= \frac{a^{\frac{b}{2}} b G_{2,2}^{2,1} \left(\begin{matrix} -\frac{b}{2} + 1 & \frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a}\right)}{2\sqrt{\pi}} \\ \frac{\left(z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b}{\sqrt{z^q p + a}} &= \frac{a^{\frac{b}{2} - \frac{1}{2}} G_{2,2}^{2,1} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2} & -\frac{b}{2} + \frac{1}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a}\right)}{\sqrt{\pi}} \\ \frac{\left(-z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^q p + a}\right)^b}{\sqrt{z^q p + a}} &= \frac{a^{\frac{b}{2} - \frac{1}{2}} G_{2,2}^{2,1} \left(\begin{matrix} -\frac{b}{2} + \frac{1}{2} & \frac{b}{2} + \frac{1}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{a}\right)}{\sqrt{\pi}} \end{aligned}$$

Functions involving $|z^q p - b|$:

$$|z^q p - b|^{-a} = 2G_{2,2}^{1,1} \left(\begin{matrix} -a + 1 & -\frac{a}{2} + \frac{1}{2} \\ 0 & -\frac{a}{2} + \frac{1}{2} \end{matrix} \middle| \frac{z^q p}{b}\right) \sin\left(\frac{\pi a}{2}\right) |b|^{-a} \Gamma(-a + 1), \text{ if } \Re(a) < 1$$

Functions involving $\text{Chi}(z^q p)$:

$$\text{Chi}(z^q p) = -\frac{\pi^{\frac{3}{2}} G_{2,4}^{2,0} \left(0, 0 \middle| \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $\text{Ci}(z^q p)$:

$$\text{Ci}(z^q p) = -\frac{\sqrt{\pi} G_{1,3}^{2,0} \left(0, 0 \middle| \frac{z^{2q} p^2}{4}\right)}{2}$$

Functions involving $\text{Ei}(z^q p)$:

$$\text{Ei}(z^q p) = -i\pi G_{1,1}^{1,0} \left(0 \middle| z\right) - G_{1,2}^{2,0} \left(0, 0 \middle| z^q p e^{i\pi}\right) - i\pi G_{1,1}^{0,1} \left(1 \middle| z\right)$$

Functions involving $\theta(z^q p - b)$:

$$\begin{aligned} (z^q p - b)^{a-1} \theta(z^q p - b) &= b^{a-1} G_{1,1}^{0,1} \left(a \middle| \frac{z^q p}{b}\right) \Gamma(a), \text{ if } b > 0 \\ (-z^q p + b)^{a-1} \theta(-z^q p + b) &= b^{a-1} G_{1,1}^{1,0} \left(0 \middle| \frac{z^q p}{b}\right) \Gamma(a), \text{ if } b > 0 \\ (z^q p - b)^{a-1} \theta\left(z - \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) &= b^{a-1} G_{1,1}^{0,1} \left(a \middle| \frac{z^q p}{b}\right) \Gamma(a), \text{ if } b > 0 \\ (-z^q p + b)^{a-1} \theta\left(-z + \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) &= b^{a-1} G_{1,1}^{1,0} \left(0 \middle| \frac{z^q p}{b}\right) \Gamma(a), \text{ if } b > 0 \end{aligned}$$

Functions involving $\theta(-z^q p + 1)$, $\log(z^q p)$:

$$\log(z^q p)^n \theta(-z^q p + 1) = \text{generated}$$

$$\log(z^q p)^n \theta(z^q p - 1) = \text{generated}$$

Functions involving $\text{Shi}(z^q p)$:

$$\text{Shi}(z^q p) = \frac{z^q \sqrt{\pi} p G_{1,3}^{1,1} \left(\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2} \middle| \frac{z^{2q} p^2 e^{i\pi}}{4} \right)}{4}$$

Functions involving $\text{Si}(z^q p)$:

$$\text{Si}(z^q p) = \frac{\sqrt{\pi} G_{1,3}^{1,1} \left(\frac{1}{2}, 0, 0 \middle| \frac{z^{2q} p^2}{4} \right)}{2}$$

Functions involving $I_a(z^q p)$:

$$I_a(z^q p) = \pi G_{1,3}^{1,0} \left(\frac{a}{2}, -\frac{a}{2}, \frac{a}{2} + \frac{1}{2} \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $J_a(z^q p)$:

$$J_a(z^q p) = G_{0,2}^{1,0} \left(\frac{a}{2}, -\frac{a}{2} \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $K_a(z^q p)$:

$$K_a(z^q p) = \frac{G_{0,2}^{2,0} \left(\frac{a}{2}, -\frac{a}{2} \middle| \frac{z^{2q} p^2}{4} \right)}{2}$$

Functions involving $Y_a(z^q p)$:

$$Y_a(z^q p) = G_{1,3}^{2,0} \left(\frac{a}{2}, -\frac{a}{2}, -\frac{a}{2} - \frac{1}{2} \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $\cos(z^q p)$:

$$\cos(z^q p) = \sqrt{\pi} G_{0,2}^{1,0} \left(0, \frac{1}{2} \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $\cosh(z^q p)$:

$$\cosh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(0, \frac{1}{2}, \frac{1}{2} \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $E(z^q p)$:

$$E(z^q p) = -\frac{G_{2,2}^{1,2} \left(\frac{1}{2}, \frac{3}{2}, 0 \middle| -z^q p \right)}{4}$$

Functions involving $K(z^q p)$:

$$K(z^q p) = \frac{G_{2,2}^{1,2} \left(\frac{1}{2}, \frac{1}{2}, 0 \middle| -z^q p \right)}{2}$$

Functions involving $\text{erf}(z^q p)$:

$$\text{erf}(z^q p) = \frac{G_{1,2}^{1,1} \left(\frac{1}{2}, 0 \middle| z^{2q} p^2 \right)}{\sqrt{\pi}}$$

Functions involving $\operatorname{erfc}(z^q p)$:

$$\operatorname{erfc}(z^q p) = \frac{G_{1,2}^{2,0} \left(0, \frac{1}{2} \middle| z^{2q} p^2 \right)}{\sqrt{\pi}}$$

Functions involving $\operatorname{erfi}(z^q p)$:

$$\operatorname{erfi}(z^q p) = \frac{z^q p G_{1,2}^{1,1} \left(\frac{1}{2}, -\frac{1}{2} \middle| -z^{2q} p^2 \right)}{\sqrt{\pi}}$$

Functions involving $e^{z^q p e^{i\pi}}$:

$$e^{z^q p e^{i\pi}} = G_{0,1}^{1,0} \left(0 \middle| z^q p \right)$$

Functions involving $E_a(z^q p)$:

$$E_a(z^q p) = G_{1,2}^{2,0} \left(a-1, 0 \middle| z^q p \right)$$

Functions involving $C(z^q p)$:

$$C(z^q p) = \frac{G_{1,3}^{1,1} \left(\frac{1}{4}, 0, \frac{3}{4} \middle| \frac{z^{4q} \pi^2 p^4}{16} \right)}{2}$$

Functions involving $S(z^q p)$:

$$S(z^q p) = \frac{G_{1,3}^{1,1} \left(\frac{3}{4}, 0, \frac{1}{4} \middle| \frac{z^{4q} \pi^2 p^4}{16} \right)}{2}$$

Functions involving $\log(z^q p)$:

$$\log(z^q p)^n = \text{generated}$$

$$\log(z^q p + a) = G_{1,1}^{1,0} \left(0 \middle| z \right) \log(a) + G_{1,1}^{0,1} \left(1 \middle| 0 \right) \log(a) + G_{2,2}^{1,2} \left(\frac{1}{1}, \frac{1}{0} \middle| \frac{z^q p}{a} \right)$$

$$\log(|z^q p - a|) = G_{1,1}^{1,0} \left(0 \middle| z \right) \log(|a|) + G_{1,1}^{0,1} \left(1 \middle| 0 \right) \log(|a|) + \pi G_{3,3}^{1,2} \left(\frac{1}{1}, \frac{1}{0}, \frac{1}{2} \middle| \frac{z^q p}{a} \right)$$

Functions involving $\sin(z^q p)$:

$$\sin(z^q p) = \sqrt{\pi} G_{0,2}^{1,0} \left(\frac{1}{2}, 0 \middle| \frac{z^{2q} p^2}{4} \right)$$

Functions involving $\operatorname{sinc}(z^q p)$:

$$\operatorname{sinc}(z^q p) = \frac{\sqrt{\pi} G_{0,2}^{1,0} \left(0, -\frac{1}{2} \middle| \frac{z^{2q} p^2}{4} \right)}{2}$$

Functions involving $\sinh(z^q p)$:

$$\sinh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(\frac{1}{2}, 1, 0 \middle| \frac{z^{2q} p^2}{4} \right)$$

5.12.4 API reference

`sympy.integrals.integrate(f, var, ...)`

Compute definite or indefinite integral of one or more variables using Risch-Norman algorithm and table lookup. This procedure is able to handle elementary algebraic and transcendental functions and also a huge class of special functions, including Airy, Bessel, Whittaker and Lambert.

var can be:

- a symbol – indefinite integration
- **a tuple (symbol, a) – indefinite integration with result** given with *a* replacing *symbol*
- a tuple (symbol, a, b) – definite integration

Several variables can be specified, in which case the result is multiple integration. (If var is omitted and the integrand is univariate, the indefinite integral in that variable will be performed.)

Indefinite integrals are returned without terms that are independent of the integration variables. (see examples)

Definite improper integrals often entail delicate convergence conditions. Pass `conds='piecewise'`, `'separate'` or `'none'` to have these returned, respectively, as a Piecewise function, as a separate result (i.e. result will be a tuple), or not at all (default is `'piecewise'`).

Strategy

SymPy uses various approaches to definite integration. One method is to find an antiderivative for the integrand, and then use the fundamental theorem of calculus. Various functions are implemented to integrate polynomial, rational and trigonometric functions, and integrands containing DiracDelta terms.

SymPy also implements the part of the Risch algorithm, which is a decision procedure for integrating elementary functions, i.e., the algorithm can either find an elementary antiderivative, or prove that one does not exist. There is also a (very successful, albeit somewhat slow) general implementation of the heuristic Risch algorithm. This algorithm will eventually be phased out as more of the full Risch algorithm is implemented. See the docstring of `Integral.eval_integral()` for more details on computing the antiderivative using algebraic methods.

The option `risch=True` can be used to use only the (full) Risch algorithm. This is useful if you want to know if an elementary function has an elementary antiderivative. If the indefinite Integral returned by this function is an instance of `NonElementaryIntegral`, that means that the Risch algorithm has proven that integral to be non-elementary. Note that by default, additional methods (such as the Meijer G method outlined below) are tried on these integrals, as they may be expressible in terms of special functions, so if you only care about elementary answers, use `risch=True`. Also note that an unevaluated Integral returned by this function is not necessarily a `NonElementaryIntegral`, even with `risch=True`, as it may just be an indication that the particular part of the Risch algorithm needed to integrate that function is not yet implemented.

Another family of strategies comes from re-writing the integrand in terms of so-called Meijer G-functions. Indefinite integrals of a single G-function can always be computed, and the definite integral of a product of two G-functions can be computed from zero to infinity. Various strategies are implemented to rewrite integrands as G-functions, and use this information to compute integrals (see the `meijerint` module).

The option `manual=True` can be used to use only an algorithm that tries to mimic integration by hand. This algorithm does not handle as many integrands as the other algorithms implemented but may return results in a more familiar form. The `manualintegrate` module has functions that return the steps used (see the module docstring for more information).

In general, the algebraic methods work best for computing antiderivatives of (possibly complicated) combinations of elementary functions. The G-function methods work best for computing definite integrals from zero to infinity of moderately complicated combinations of special functions, or indefinite integrals of very simple combinations of special functions.

The strategy employed by the integration code is as follows:

- If computing a definite integral, and both limits are real, and at least one limit is $+\infty$, try the G-function method of definite integration first.
- Try to find an antiderivative, using all available methods, ordered by performance (that is try fastest method first, slowest last; in particular polynomial integration is tried first, Meijer G-functions second to last, and heuristic Risch last).
- If still not successful, try G-functions irrespective of the limits.

The option `meijerg=True, False, None` can be used to, respectively: always use G-function methods and no others, never use G-function methods, or use all available methods (in order as described above). It defaults to `None`.

See also:

[Integral](#) (page 690), [Integral.doit](#) (page 692)

Examples

```
>>> from sympy import integrate, log, exp, oo
>>> from sympy.abc import a, x, y
```

```
>>> integrate(x*y, x)
x**2*y/2
```

```
>>> integrate(log(x), x)
x*log(x) - x
```

```
>>> integrate(log(x), (x, 1, a))
a*log(a) - a + 1
```

```
>>> integrate(x)
x**2/2
```

Terms that are independent of x are dropped by indefinite integration:

```
>>> from sympy import sqrt
>>> integrate(sqrt(1 + x), (x, 0, x))
2*(x + 1)**(3/2)/3 - 2/3
>>> integrate(sqrt(1 + x), x)
2*(x + 1)**(3/2)/3
```

```
>>> integrate(x*y)
Traceback (most recent call last):
...
ValueError: specify integration variables to integrate x*y
```

Note that `integrate(x)` syntax is meant only for convenience in interactive sessions and should be avoided in library code.

```
>>> integrate(x**a*exp(-x), (x, 0, oo)) # same as conds='piecewise'
Piecewise((gamma(a + 1), -re(a) < 1),
          (Integral(x**a*exp(-x), (x, 0, oo)), True))
```

```
>>> integrate(x**a*exp(-x), (x, 0, oo), conds='none')
gamma(a + 1)
```

```
>>> integrate(x**a*exp(-x), (x, 0, oo), conds='separate')
(gamma(a + 1), -re(a) < 1)
```

`sympy.integrals.line_integrate`(field, Curve, variables)

Compute the line integral.

See also:

[integrate](#) (page 688), [Integral](#) (page 690)

Examples

```
>>> from sympy import Curve, line_integrate, E, ln
>>> from sympy.abc import x, y, t
>>> C = Curve([E**t + 1, E**t - 1], (t, 0, ln(2)))
>>> line_integrate(x + y, C, [x, y])
3*sqrt(2)
```

The class *Integral* represents an unevaluated integral and has some methods that help in the integration of an expression.

class `sympy.integrals.Integral`

Represents unevaluated integral.

is_commutative

Returns whether all the free symbols in the integral are commutative.

as_sum(n=None, method='midpoint', evaluate=True)

Approximates a definite integral by a sum.

See also:

[Integral.doit](#) (page 692) Perform the integration using any hints

Examples

```
>>> from sympy import sin, sqrt
>>> from sympy.abc import x, n
>>> from sympy.integrals import Integral
>>> e = Integral(sin(x), (x, 3, 7))
```

(continues on next page)

(continued from previous page)

```
>>> e
Integral(sin(x), (x, 3, 7))
```

For demonstration purposes, this interval will only be split into 2 regions, bounded by [3, 5] and [5, 7].

The left-hand rule uses function evaluations at the left of each interval:

```
>>> e.as_sum(2, 'left')
2*sin(5) + 2*sin(3)
```

The midpoint rule uses evaluations at the center of each interval:

```
>>> e.as_sum(2, 'midpoint')
2*sin(4) + 2*sin(6)
```

The right-hand rule uses function evaluations at the right of each interval:

```
>>> e.as_sum(2, 'right')
2*sin(5) + 2*sin(7)
```

The trapezoid rule uses function evaluations on both sides of the intervals. This is equivalent to taking the average of the left and right hand rule results:

```
>>> e.as_sum(2, 'trapezoid')
2*sin(5) + sin(3) + sin(7)
>>> (e.as_sum(2, 'left') + e.as_sum(2, 'right'))/2 == _
True
```

Here, the discontinuity at $x = 0$ can be avoided by using the midpoint or right-hand method:

```
>>> e = Integral(1/sqrt(x), (x, 0, 1))
>>> e.as_sum(5).n(4)
1.730
>>> e.as_sum(10).n(4)
1.809
>>> e.doit().n(4) # the actual value is 2
2.000
```

The left- or trapezoid method will encounter the discontinuity and return infinity:

```
>>> e.as_sum(5, 'left')
zoo
```

The number of intervals can be symbolic. If omitted, a dummy symbol will be used for it. `>>> e = Integral(x**2, (x, 0, 2)) >>> e.as_sum(n, 'right').expand()` $8/3 + 4/n + 4/(3*n**2)$

This shows that the midpoint rule is more accurate, as its error term decays as the square of n : `>>> e.as_sum(method='midpoint').expand()` $8/3 - 2/(3*n**2)$

A symbolic sum is returned with `evaluate=False`: `>>> e.as_sum(n, 'midpoint', evaluate=False)` $2*\text{Sum}((2*_k/n - 1/n)**2, (_k, 1, n))/n$

Arguments

n The number of subintervals to use, optional.

method One of: 'left', 'right', 'midpoint', 'trapezoid'.

evaluate If False, returns an unevaluated Sum expression. The default is True, evaluate the sum.

These methods of approximate integration are described in [1].

[1] https://en.wikipedia.org/wiki/Riemann_sum#Methods

doit(hints)**

Perform the integration using any hints given.

See also:

`sympy.integrals.trigonometry.trigintegrate`, `sympy.integrals.risch.heurisch`, `sympy.integrals.rationaltools.ratint`

as_sum (page 690) Approximate the integral using a sum

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).doit()
Piecewise((x**3/log(x) - x/log(x),
          (x > 1) | ((x >= 0) & (x < 1))), (2, True))
```

free_symbols

This method returns the symbols that will exist when the integral is evaluated. This is useful if one is trying to determine whether an integral depends on a certain symbol or not.

See also:

`function`, `limits`, `variables`

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x, (x, y, 1)).free_symbols
{y}
```

transform(x, u)

Performs a change of variables from x to u using the relationship given by x and u which will define the transformations f and F (which are inverses of each other) as follows:

1. If x is a Symbol (which is a variable of integration) then u will be interpreted as some function, $f(u)$, with inverse $F(u)$. This, in effect, just makes the substitution of x with $f(x)$.
2. If u is a Symbol then x will be interpreted as some function, $F(x)$, with inverse $f(u)$. This is commonly referred to as u -substitution.

Once f and F have been identified, the transformation is made as follows:

$$\int_a^b x dx \rightarrow \int_{F(a)}^{F(b)} f(x) \frac{d}{dx}$$

where $F(x)$ is the inverse of $f(x)$ and the limits and integrand have been corrected so as to retain the same value after integration.

See also:

variables Lists the integration variables

as_dummy Replace integration variables with dummy ones

Notes

The mappings, $F(x)$ or $f(u)$, must lead to a unique integral. Linear or rational linear expression, $2*x$, $1/x$ and $\text{sqrt}(x)$, will always work; quadratic expressions like $x**2 - 1$ are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if x is not a variable of integration.

x must be (or contain) only one of the integration variables. If u has more than one free symbol then it should be sent as a tuple $(u, uvar)$ where $uvar$ identifies which variable is replacing the integration variable. XXX can it contain another integration variable?

Examples

```
>>> from sympy.abc import a, b, c, d, x, u, y
>>> from sympy import Integral, S, cos, sqrt
```

```
>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))
```

transform can change the variable of integration

```
>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))
```

transform can perform u-substitution as long as a unique integrand is obtained:

```
>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```

This attempt fails because $x = \pm\sqrt{u + 1}$ and the sign does not cancel out of the integrand:

```
>>> Integral(cos(x**2 - 1), (x, 0, 1)).transform(x**2 - 1, u)
Traceback (most recent call last):
...
ValueError:
The mapping between F(x) and f(u) did not give a unique integrand.
```

transform can do a substitution. Here, the previous result is transformed back into the original expression using “u-substitution”:

```
>>> ui = _
>>> _.transform(sqrt(u + 1), x) == i
True
```

We can accomplish the same with a regular substitution:

```
>>> ui.transform(u, x**2 - 1) == i
True
```

If the x does not contain a symbol of integration then the integral will be returned unchanged. Integral i does not have an integration variable a so no change is made:

```
>>> i.transform(a, x) == i
True
```

When u has more than one free symbol the symbol that is replacing x must be identified by passing u as a tuple:

```
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, u))
Integral(a + u, (u, -a, -a + 1))
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, a))
Integral(a + u, (a, -u, -u + 1))
```

5.12.5 TODO and Bugs

There are still lots of functions that SymPy does not know how to integrate. For bugs related to this module, see <https://github.com/sympy/sympy/issues?q=label%3AIntegration>

5.13 Numeric Integrals

SymPy has functions to calculate points and weights for Gaussian quadrature of any order and any precision:

`sympy.integrals.quadrature.gauss_legendre(n, n_digits)`

Computes the Gauss-Legendre quadrature [R373] (page 1865) points and weights.

The Gauss-Legendre quadrature approximates the integral:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of P_n and the weights w_i are given by:

$$w_i = \frac{2}{(1 - x_i^2) (P'_n(x_i))^2}$$

Parameters **n** : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns **(x, w)** : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_laguerre](#) (page 695), [gauss_gen_laguerre](#) (page 697), [gauss_hermite](#) (page 696), [gauss_chebyshev_t](#) (page 697), [gauss_chebyshev_u](#) (page 698), [gauss_jacobi](#) (page 699), [gauss_lobatto](#)

References

[R373] (page 1865), [R374] (page 1865)

Examples

```
>>> from sympy.integrals.quadrature import gauss_legendre
>>> x, w = gauss_legendre(3, 5)
>>> x
[-0.7746, 0, 0.7746]
>>> w
[0.55556, 0.88889, 0.55556]
>>> x, w = gauss_legendre(4, 5)
>>> x
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> w
[0.34785, 0.65215, 0.65215, 0.34785]
```

`sympy.integrals.quadrature.gauss_laguerre(n, n_digits)`

Computes the Gauss-Laguerre quadrature [R375] (page 1865) points and weights.

The Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n and the weights w_i are given by:

$$w_i = \frac{x_i}{(n+1)^2 (L_{n+1}(x_i))^2}$$

Parameters **n** : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (**x**, **w**) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_legendre](#) (page 694), [gauss_gen_laguerre](#) (page 697), [gauss_hermite](#) (page 696), [gauss_chebyshev_t](#) (page 697), [gauss_chebyshev_u](#) (page 698), [gauss_jacobi](#) (page 699), [gauss_lobatto](#)

References

[R375] (page 1865), [R376] (page 1865)

Examples

```
>>> from sympy.integrals.quadrature import gauss_laguerre
>>> x, w = gauss_laguerre(3, 5)
>>> x
[0.41577, 2.2943, 6.2899]
>>> w
[0.71109, 0.27852, 0.010389]
>>> x, w = gauss_laguerre(6, 5)
>>> x
[0.22285, 1.1889, 2.9927, 5.7751, 9.8375, 15.983]
>>> w
[0.45896, 0.417, 0.11337, 0.010399, 0.00026102, 8.9855e-7]
```

`sympy.integrals.quadrature.gauss_hermite(n, n_digits)`

Computes the Gauss-Hermite quadrature [\[R377\]](#) (page 1865) points and weights.

The Gauss-Hermite quadrature approximates the integral:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of H_n and the weights w_i are given by:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 (H_{n-1}(x_i))^2}$$

Parameters **n** : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_legendre](#) (page 694), [gauss_laguerre](#) (page 695), [gauss_gen_laguerre](#) (page 697), [gauss_chebyshev_t](#) (page 697), [gauss_chebyshev_u](#) (page 698), [gauss_jacobi](#) (page 699), [gauss_lobatto](#)

References

[\[R377\]](#) (page 1865), [\[R378\]](#) (page 1865), [\[R379\]](#) (page 1865)

Examples

```
>>> from sympy.integrals.quadrature import gauss_hermite
>>> x, w = gauss_hermite(3, 5)
>>> x
[-1.2247, 0, 1.2247]
>>> w
[0.29541, 1.1816, 0.29541]
```

```
>>> x, w = gauss_hermite(6, 5)
>>> x
[-2.3506, -1.3358, -0.43608, 0.43608, 1.3358, 2.3506]
>>> w
[0.00453, 0.15707, 0.72463, 0.72463, 0.15707, 0.00453]
```

`sympy.integrals.quadrature.gauss_gen_laguerre(n, alpha, n_digits)`

Computes the generalized Gauss-Laguerre quadrature [R380] (page 1865) points and weights.

The generalized Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} x^{\alpha} e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n^{α} and the weights w_i are given by:

$$w_i = \frac{\Gamma(\alpha + n)}{n\Gamma(n)L_{n-1}^{\alpha}(x_i)L_{n-1}^{\alpha+1}(x_i)}$$

Parameters **n** : the order of quadrature

alpha : the exponent of the singularity, $\alpha > -1$

n_digits : number of significant digits of the points and weights to return

Returns **(x, w)** : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_legendre](#) (page 694), [gauss_laguerre](#) (page 695), [gauss_hermite](#) (page 696), [gauss_chebyshev_t](#) (page 697), [gauss_chebyshev_u](#) (page 698), [gauss_jacobi](#) (page 699), [gauss_lobatto](#)

References

[R380] (page 1865), [R381] (page 1865)

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_gen_laguerre
>>> x, w = gauss_gen_laguerre(3, -S.Half, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> w
[1.4493, 0.31413, 0.00906]
```

```
>>> x, w = gauss_gen_laguerre(4, 3*S.Half, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

`sympy.integrals.quadrature.gauss_chebyshev_t(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R382] (page 1865) points and weights of the first kind.

The Gauss-Chebyshev quadrature of the first kind approximates the integral:

$$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of T_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n}$$

Parameters **n** : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_legendre](#) (page 694), [gauss_laguerre](#) (page 695), [gauss_hermite](#) (page 696), [gauss_gen_laguerre](#) (page 697), [gauss_chebyshev_u](#) (page 698), [gauss_jacobi](#) (page 699), [gauss_lobatto](#)

References

[R382] (page 1865), [R383] (page 1865)

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_chebyshev_t
>>> x, w = gauss_chebyshev_t(3, 5)
>>> x
[0.86602, 0, -0.86602]
>>> w
[1.0472, 1.0472, 1.0472]
```

```
>>> x, w = gauss_chebyshev_t(6, 5)
>>> x
[0.96593, 0.70711, 0.25882, -0.25882, -0.70711, -0.96593]
>>> w
[0.5236, 0.5236, 0.5236, 0.5236, 0.5236, 0.5236]
```

`sympy.integrals.quadrature.gauss_chebyshev_u(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R384] (page 1865) points and weights of the second kind.

The Gauss-Chebyshev quadrature of the second kind approximates the integral:

$$\int_{-1}^1 \sqrt{1-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of U_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n+1} \sin^2 \left(\frac{i}{n+1} \pi \right)$$

Parameters **n** : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (**x**, **w**) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_legendre](#) (page 694), [gauss_laguerre](#) (page 695), [gauss_hermite](#) (page 696), [gauss_gen_laguerre](#) (page 697), [gauss_chebyshev_t](#) (page 697), [gauss_jacobi](#) (page 699), [gauss_lobatto](#)

References

[\[R384\]](#) (page 1865), [\[R385\]](#) (page 1865)

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_chebyshev_u
>>> x, w = gauss_chebyshev_u(3, 5)
>>> x
[0.70711, 0, -0.70711]
>>> w
[0.3927, 0.7854, 0.3927]
```

```
>>> x, w = gauss_chebyshev_u(6, 5)
>>> x
[0.90097, 0.62349, 0.22252, -0.22252, -0.62349, -0.90097]
>>> w
[0.084489, 0.27433, 0.42658, 0.42658, 0.27433, 0.084489]
```

`sympy.integrals.quadrature.gauss_jacobi(n, alpha, beta, n_digits)`
Computes the Gauss-Jacobi quadrature [\[R386\]](#) (page 1865) points and weights.

The Gauss-Jacobi quadrature of the first kind approximates the integral:

$$\int_{-1}^1 (1-x)^\alpha (1+x)^\beta f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of $P_n^{(\alpha, \beta)}$ and the weights w_i are given by:

$$w_i = -\frac{2n + \alpha + \beta + 2}{n + \alpha + \beta + 1} \frac{\Gamma(n + \alpha + 1) \Gamma(n + \beta + 1)}{\Gamma(n + \alpha + \beta + 1) (n + 1)!} \frac{2^{\alpha + \beta}}{P'_n(x_i) P_{n+1}^{(\alpha, \beta)}(x_i)}$$

Parameters **n** : the order of quadrature

alpha : the first parameter of the Jacobi Polynomial, $\alpha > -1$

beta : the second parameter of the Jacobi Polynomial, $\beta > -1$

n_digits : number of significant digits of the points and weights to return

Returns (**x**, **w**) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

[gauss_legendre](#) (page 694), [gauss_laguerre](#) (page 695), [gauss_hermite](#) (page 696), [gauss_gen_laguerre](#) (page 697), [gauss_chebyshev_t](#) (page 697), [gauss_chebyshev_u](#) (page 698), [gauss_lobatto](#)

References

[R386] (page 1865), [R387] (page 1865), [R388] (page 1865)

Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_jacobi
>>> x, w = gauss_jacobi(3, S.Half, -S.Half, 5)
>>> x
[-0.90097, -0.22252, 0.62349]
>>> w
[1.7063, 1.0973, 0.33795]
```

```
>>> x, w = gauss_jacobi(6, 1, 1, 5)
>>> x
[-0.87174, -0.5917, -0.2093, 0.2093, 0.5917, 0.87174]
>>> w
[0.050584, 0.22169, 0.39439, 0.39439, 0.22169, 0.050584]
```

5.14 Integration over Polytopes

The `intpoly` module in SymPy implements methods to calculate the integral of a polynomial over 2/3-Polytopes. Uses evaluation techniques as described in Chin et al. (2015) [1].

The input for 2-Polytope or Polygon uses the already existing Polygon data structure in SymPy. See [sympy.geometry.polygon](#) (page 627) for how to create a polygon.

For the 3-Polytope or Polyhedron, the most economical representation is to specify a list of vertices and then to provide each constituting face(Polygon) as a list of vertex indices.

For example, consider the unit cube. Here is how it would be represented.

```
unit_cube = [[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0),
              [3, 7, 6, 2], [1, 5, 7, 3], [5, 4, 6, 7], [0, 4, 5, 1], [2, 0, 1, 3], [2,
              6, 4, 0]]
```

Here, the first sublist is the list of vertices. The other smaller lists such as [3, 7, 6, 2] represent a 2D face of the polyhedra with vertices having index 3, 7, 6 and 2 in the first sublist(in that order).

Principal method in this module is `polytope_integrate()` (page 702)

- `polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x)` returns the integral of x over the triangle with vertices (0, 0), (0, 1) and (1, 0)
- `polytope_integrate(unit_cube, x + y + z)` returns the integral of $x + y + z$ over the unit cube.

5.14.1 References

[1]: Chin, Eric B., Jean B. Lasserre, and N. Sukumar. "Numerical integration of homogeneous functions on convex and nonconvex polygons and polyhedra." Computational Mechanics 56.6 (2015): 967-981

PDF link : <http://dilbert.engr.ucdavis.edu/~suku/quadrature/cls-integration.pdf>

5.14.2 Examples

5.14.3 For 2D Polygons

Single Polynomial:

```
>>> from sympy.integrals.intpoly import *
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x)
1/6
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x + x*y + y**2)
7/24
```

List of specified polynomials:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4], max_
↳ degree=4)
          4          2
{3: 3/2, x : 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.89*x**3,
↳ x*y + y**2, x**4], max_degree=4)
          2          3 689      4          2
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x : ----, x : 1/30, x*y + y : 1/8}
          2000
```

Computing all monomials up to a maximum degree:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), max_degree=3)
          2          3          2          3
↳ 2          2
{0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/24, x*y
↳ : 1/60, x *y: 1/60}
```

5.14.4 For 3-Polytopes/Polyhedra

Single Polynomial:

```
>>> from sympy.integrals.intpoly import *
>>> cube = [[(0, 0, 0), (0, 0, 5), (0, 5, 0), (0, 5, 5), (5, 0, 0), (5, 0, 5), (5, 5, 0), (5, 5, 5)], [2, 6, 7, 3], [3, 7, 5, 1], [7, 6, 4, 5], [1, 5, 4, 0], [3, 1, 0, 2], [0, 4, 6, 2]]
>>> polytope_integrate(cube, x**2 + y**2 + z**2 + x*y + y*z + x*z)
-21875/4
>>> octahedron = [(S(-1) / sqrt(2), 0, 0), (0, S(1) / sqrt(2), 0), (0, 0, S(-1) / sqrt(2)), (0, 0, S(1) / sqrt(2)), (0, S(-1) / sqrt(2), 0), (S(1) / sqrt(2), 0, 0)], [3, 4, 5], [3, 5, 1], [3, 1, 0], [3, 0, 4], [4, 0, 2], [4, 2, 5], [2, 0, 1], [5, 2, 1]]
>>> polytope_integrate(octahedron, x**2 + y**2 + z**2 + x*y + y*z + x*z)

$$\frac{\sqrt{2}}{20}$$

```

List of specified polynomials:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4], max_degree=4)
{3: 3/2, x : 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.89*x**3, x*y + y**2, x**4], max_degree=4)
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x :  $\frac{3 \cdot 689}{2000}$ , x : 1/30, x*y + y : 1/8}
```

Computing all monomials up to a maximum degree:

```
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), max_degree=3)
{0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/24, x*y : 1/60, x *y: 1/60}
```

5.14.5 API reference

`sympy.integrals.intpoly.polytope_integrate`(poly, expr=None, **kwargs)
Integrates polynomials over 2/3-Polytopes.

This function accepts the polytope in *poly* and the function in *expr* (uni/bi/trivariate polynomials are implemented) and returns the exact integral of *expr* over *poly*.

Parameters *poly* : The input Polygon.

expr : The input polynomial.

clockwise : Binary value to sort input points of 2-Polytope clockwise.(Optional)

max_degree : The maximum degree of any monomial of the input polynomial.(Optional)

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.geometry.polygon import Polygon
>>> from sympy.geometry.point import Point
>>> from sympy.integrals.intpoly import polytope_integrate
>>> polygon = Polygon(Point(0,0), Point(0,1), Point(1,1), Point(1,0))
>>> polys = [1, x, y, x*y, x**2*y, x*y**2]
>>> expr = x*y
>>> polytope_integrate(polygon, expr)
1/4
>>> polytope_integrate(polygon, polys, max_degree=3)
{1: 1, x: 1/2, y: 1/2, x*y: 1/4, x*y**2: 1/6, x**2*y: 1/6}
```

5.15 Lie Algebra Module

class sympy.liealgebras.root_system.RootSystem

Represent the root system of a simple Lie algebra

Every simple Lie algebra has a unique root system. To find the root system, we first consider the Cartan subalgebra of $\mathfrak{g}$, which is the maximal abelian subalgebra, and consider the adjoint action of $\mathfrak{g}$ on this subalgebra. There is a root system associated with this action. Now, a root system over a vector space V is a set of finite vectors Φ (called roots), which satisfy:

1. The roots span V
2. The only scalar multiples of x in Φ are x and $-x$
3. For every x in Φ , the set Φ is closed under reflection through the hyperplane perpendicular to x .
4. If x and y are roots in Φ , then the projection of y onto the line through x is a half-integral multiple of x .

Now, there is a subset of Φ , which we will call Δ , such that: 1. Δ is a basis of V 2. Each root x in Φ can be written $x = \sum k_y y$ for y in Δ

The elements of Δ are called the simple roots. Therefore, we see that the simple roots span the root space of a given simple Lie algebra.

References: https://en.wikipedia.org/wiki/Root_system Lie Algebras and Representation Theory - Humphreys

add_as_root2(root1, root2)

Add two roots together if and only if their sum is also a root

It takes as input two vectors which should be roots. It then computes their sum and checks if it is in the list of all possible roots. If it is, it returns the sum. Otherwise it returns a string saying that the sum is not a root.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
```

(continues on next page)

(continued from previous page)

```
>>> c.add_as_roots([1, 0, -1, 0], [0, 0, 1, -1])
[1, 0, 0, -1]
>>> c.add_as_roots([1, -1, 0, 0], [0, 0, -1, 1])
'The sum of these two roots is not a root'
```

add_simple_roots(root1, root2)

Add two simple roots together

The function takes as input two integers, root1 and root2. It then uses these integers as keys in the dictionary of simple roots, and gets the corresponding simple roots, and then adds them together.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> newroot = c.add_simple_roots(1, 2)
>>> newroot
[1, 0, -1, 0]
```

all_roots()

Generate all the roots of a given root system

The result is a dictionary where the keys are integer numbers. It generates the roots by getting the dictionary of all positive roots from the bases classes, and then taking each root, and multiplying it by -1 and adding it to the dictionary. In this way all the negative roots are generated.

cartan_matrix()

Cartan matrix of Lie algebra associated with this root system

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.cartan_matrix()
Matrix([
  [ 2, -1,  0],
  [-1,  2, -1],
  [ 0, -1,  2]])
```

dynkin_diagram()

Dynkin diagram of the Lie algebra associated with this root system

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> print(c.dynkin_diagram())
0---0---0
1  2  3
```

root_space()

Return the span of the simple roots

The root space is the vector space spanned by the simple roots, i.e. it is a vector space with a distinguished basis, the simple roots. This method returns a string that represents the root space as the span of the simple roots, $\alpha[1], \dots, \alpha[n]$.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.root_space()
'alpha[1] + alpha[2] + alpha[3]'
```

simple_roots()

Generate the simple roots of the Lie algebra

The rank of the Lie algebra determines the number of simple roots that it has. This method obtains the rank of the Lie algebra, and then uses the `simple_root` method from the Lie algebra classes to generate all the simple roots.

Examples

```
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> roots = c.simple_roots()
>>> roots
{1: [1, -1, 0, 0], 2: [0, 1, -1, 0], 3: [0, 0, 1, -1]}
```

class sympy.liealgebras.type_a.TypeA

This class contains the information about the A series of simple Lie algebras. ===

basic_root(i, j)

This is a method just to generate roots with a 1 in the i th position and a -1 in the j th position.

basis()

Returns the number of independent generators of A_n

cartan_matrix()

Returns the Cartan matrix for A_n . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots, $(\alpha[1], \dots, \alpha[l])$. Then the ij th entry of the Cartan matrix is $(\langle \alpha[i], \alpha[j] \rangle)$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.dimension()
5
```

highest_root()

Returns the highest weight root for A_n

lie_algebra()

Returns the Lie algebra associated with A_n

positive_roots()

This method generates all the positive roots of A_n . This is half of all of the roots of A_n ; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for A_n

simple_root(i)

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In A_n the i th simple root is the root which has a 1 in the i th position, a -1 in the $(i+1)$ th position, and zeroes elsewhere.

This method returns the i th simple root for the A series.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.simple_root(1)
[1, -1, 0, 0, 0]
```

class sympy.liealgebras.type_b.TypeB

basic_root(i, j)

This is a method just to generate roots with a 1 in the i th position and a -1 in the j th position.

basis()

Returns the number of independent generators of B_n

cartan_matrix()

Returns the Cartan matrix for B_n . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots, $(\alpha[1], \dots, \alpha[l])$. Then the ij th entry of the Cartan matrix is $\langle \alpha[i], \alpha[j] \rangle$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('B4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -2],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("B3")
>>> c.dimension()
3
```

lie_algebra()

Returns the Lie algebra associated with B_n

positive_roots()

This method generates all the positive roots of A_n . This is half of all of the roots of B_n ; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for B_n

simple_root(i)

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written

as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In B_n the first $n-1$ simple roots are the same as the roots in $A_{(n-1)}$ (a 1 in the i th position, a -1 in the $(i+1)$ th position, and zeroes elsewhere). The n -th simple root is the root with a 1 in the n th position and zeroes elsewhere.

This method returns the i th simple root for the B series.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("B3")
>>> c.simple_root(2)
[0, 1, -1]
```

class sympy.liealgebras.type_c.TypeC

basic_root(i, j)

Generate roots with 1 in i th position and a -1 in j th position

basis()

Returns the number of independent generators of C_n

cartan_matrix()

The Cartan matrix for C_n

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, ($\alpha[1], \dots, \alpha[l]$). Then the ij th entry of the Cartan matrix is $\langle \alpha[i], \alpha[j] \rangle$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('C4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -2,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.dimension()
3
```

lie_algebra()

Returns the Lie algebra associated with C_n

positive_roots()

Generates all the positive roots of A_n

This is half of all of the roots of C_n ; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for C_n

simple_root(i)

The i th simple root for the C series

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In C_n , the first $n-1$ simple roots are the same as the roots in $A_{(n-1)}$ (a 1 in the i th position, a -1 in the $(i+1)$ th position, and zeroes elsewhere). The n th simple root is the root in which there is a 2 in the n th position and zeroes elsewhere.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.simple_root(2)
[0, 1, -1]
```

class sympy.liealgebras.type_d.TypeD**basic_root(i, j)**

This is a method just to generate roots with a 1 in the i th position and a -1 in the j th position.

basis()

Returns the number of independent generators of D_n

cartan_matrix()

Returns the Cartan matrix for D_n . The Cartan matrix for a Lie algebra is generated by assigning an ordering to the simple roots, $(\alpha[1], \dots, \alpha[l])$. Then the ij th entry of the Cartan matrix is $\langle \alpha[i], \alpha[j] \rangle$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('D4')
>>> c.cartan_matrix()
Matrix([
  [ 2, -1,  0,  0],
  [-1,  2, -1, -1],
  [ 0, -1,  2,  0],
  [ 0, -1,  0,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.dimension()
4
```

lie_algebra()

Returns the Lie algebra associated with D_n

positive_roots()

This method generates all the positive roots of A_n . This is half of all of the roots of D_n by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for D_n

simple_root(i)

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In D_n , the first $n-1$ simple roots are the same as the roots in $A_{(n-1)}$ (a 1 in the i th position, a -1 in the $(i+1)$ th position, and zeroes elsewhere). The n th simple root is the root in which there 1s in the n th and $(n-1)$ th positions, and zeroes elsewhere.

This method returns the i th simple root for the D series.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.simple_root(2)
[0, 1, -1, 0]
```

class sympy.liealgebras.type_e.TypeE

basic_root(i, j)

This is a method just to generate roots with a -1 in the ith position and a 1 in the jth position.

basis()

Returns the number of independent generators of E_n

cartan_matrix()

Returns the Cartan matrix for G_2 The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, ($\alpha[1], \dots, \alpha[l]$). Then the ij th entry of the Cartan matrix is $(\langle \alpha[i], \alpha[j] \rangle)$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.dimension()
8
```

positive_roots()

This method generates all the positive roots of A_n . This is half of all of the roots of E_n ; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
```

(continues on next page)

(continued from previous page)

```
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots of E_n

simple_root(i)

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

This method returns the i th simple root for E_n .

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.simple_root(2)
[1, 1, 0, 0, 0, 0, 0, 0]
```

class sympy.liealgebras.type_f.TypeF**basic_root(i, j)**

Generate roots with 1 in i th position and -1 in j th position

basis()

Returns the number of independent generators of F_4

cartan_matrix()

The Cartan matrix for F_4

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, ($\alpha[1]$, ..., $\alpha[l]$). Then the ij th entry of the Cartan matrix is $\langle \alpha[i], \alpha[j] \rangle$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([
[ 2, -1,  0,  0],
[-1,  2, -1,  0],
[ 0, -1,  2, -1],
[ 0,  0, -1,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.dimension()
4
```

positive_roots()

Generate all the positive roots of A_n

This is half of all of the roots of F_4 ; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots for F_4

simple_root(i)

The i th simple root of F_4

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.simple_root(3)
[0, 0, 0, 1]
```

class sympy.liealgebras.type_g.TypeG

basis()

Returns the number of independent generators of G_2

cartan_matrix()

The Cartan matrix for G_2

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, ($\alpha[1]$, ..., $\alpha[l]$). Then the ij th entry of the Cartan matrix is $\langle \alpha[i], \alpha[j] \rangle$.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.cartan_matrix()
Matrix([
  [ 2, -1],
  [-3,  2]])
```

dimension()

Dimension of the vector space V underlying the Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.dimension()
3
```

positive_roots()

Generate all the positive roots of A_n

This is half of all of the roots of A_n ; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()

Returns the total number of roots of G_2

simple_root(i)

The i th simple root of G_2

Every lie algebra has a unique root system. Given a root system Q , there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q . If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

Examples

```
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.simple_root(1)
[0, 1, -1]
```


class sympy.liealgebras.weyl_group.WeylGroup

For each semisimple Lie group, we have a Weyl group. It is a subgroup of the isometry group of the root system. Specifically, it's the subgroup that is generated by reflections through the hyperplanes orthogonal to the roots. Therefore, Weyl groups are reflection groups, and so a Weyl group is a finite Coxeter group.

coxeter_diagram()

This method returns the Coxeter diagram corresponding to a Weyl group. The Coxeter diagram can be obtained from a Lie algebra's Dynkin diagram by deleting all arrows; the Coxeter diagram is the undirected graph. The vertices of the Coxeter diagram represent the generating reflections of the Weyl group, s_i . An edge is drawn between s_i and s_j if the order $m(i, j)$ of $s_i s_j$ is greater than two. If there is one edge, the order $m(i, j)$ is 3. If there are two edges, the order $m(i, j)$ is 4, and if there are three edges, the order $m(i, j)$ is 6.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("B3")
>>> print(c.coxeter_diagram())
0--0==0
1  2  3
```

delete_doubles(reflections)

This is a helper method for determining the order of an element in the Weyl group of G_2 . It takes a Weyl element and if repeated simple reflections in it, it deletes them.

element_order(weylelt)

This method returns the order of a given Weyl group element, which should be specified by the user in the form of products of the generating reflections, i.e. of the form $r_1 r_2$ etc.

For types A-F, this method current works by taking the matrix form of the specified element, and then finding what power of the matrix is the identity. It then returns this power.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> b = WeylGroup("B4")
>>> b.element_order('r1*r4*r2')
4
```

generators()

This method creates the generating reflections of the Weyl group for a given Lie algebra. For a Lie algebra of rank n , there are n different generating reflections. This function returns them as a list.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("F4")
>>> c.generators()
['r1', 'r2', 'r3', 'r4']
```

group_name()

This method returns some general information about the Weyl group for a given Lie algebra. It returns the name of the group and the elements it acts on, if relevant.

group_order()

This method returns the order of the Weyl group. For types A, B, C, D, and E the order depends on the rank of the Lie algebra. For types F and G, the order is fixed.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("D4")
>>> c.group_order()
192.0
```

matrix_form(weylelt)

This method takes input from the user in the form of products of the generating reflections, and returns the matrix corresponding to the element of the Weyl group. Since each element of the Weyl group is a reflection of some type, there is a corresponding matrix representation. This method uses the standard representation for all the generating reflections.

Examples

```
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> f = WeylGroup("F4")
>>> f.matrix_form('r2*r3')
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 0, -1],
[0, 0, 1, 0]])
```

class sympy.liealgebras.cartan_type.**CartanType_generator**
Constructor for actually creating things

class sympy.liealgebras.cartan_type.**Standard_Cartan**
Concrete base class for Cartan types such as A4, etc

rank()

Returns the rank of the Lie algebra

series()

Returns the type of the Lie algebra

sympy.liealgebras.dynkin_diagram.**DynkinDiagram(t)**

Display the Dynkin diagram of a given Lie algebra

Works by generating the CartanType for the input, t, and then returning the Dynkin diagram method from the individual classes.

Examples

```
>>> from sympy.liealgebras.dynkin_diagram import DynkinDiagram
>>> print(DynkinDiagram("A3"))
0---0---0
1   2   3
```

```
>>> print(DynkinDiagram("B4"))
0---0---0==>0
1   2   3   4
```

`sympy.liealgebras.cartan_matrix.CartanMatrix(ct)`
Access the Cartan matrix of a specific Lie algebra

Examples

```
>>> from sympy.liealgebras.cartan_matrix import CartanMatrix
>>> CartanMatrix("A2")
Matrix([
[ 2, -1],
[-1,  2]])
```

```
>>> CartanMatrix(['C', 3])
Matrix([
[ 2, -1,  0],
[-1,  2, -1],
[ 0, -2,  2]])
```

This method works by returning the Cartan matrix which corresponds to Cartan type `t`.

5.16 Logic Module

5.16.1 Introduction

The logic module for SymPy allows to form and manipulate logic expressions using symbolic and Boolean values.

5.16.2 Forming logical expressions

You can build Boolean expressions with the standard python operators `&` (And), `|` (Or), `~` (Not):

```
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> y | (x & y)
y | (x & y)
>>> x | y
x | y
>>> ~x
~x
```

You can also form implications with `>>` and `<<`:

```
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in SymPy, Boolean expressions inherit from `Basic`:

```
>>> (y & x).subs({x: True, y: True})
True
>>> (x | y).atoms()
{x, y}
```

The logic module also includes the following functions to derive boolean expressions from their truth tables-

`sympy.logic.boolalg.SOPform(variables, minterms, dontcares=None)`

The `SOPform` function uses `simplified_pairs` and a redundant group-eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest Sum of Products form.

The variables must be given as the first argument.

Return a logical Or function (i.e., the “sum of products” or “SOP” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

References

[R389] (page 1865)

Examples

```
>>> from sympy.logic import SOPform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1],
...             [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (z & ~w)
```

`sympy.logic.boolalg.POSform(variables, minterms, dontcares=None)`

The `POSform` function uses `simplified_pairs` and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest Product of Sums form.

The variables must be given as the first argument.

Return a logical And function (i.e., the “product of sums” or “POS” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

References

[R390] (page 1865)

Examples

```
>>> from sympy.logic import POSform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1],
...             [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

5.16.3 Boolean functions

class sympy.logic.boolalg.BooleanTrue

SymPy version of True, a singleton that can be accessed via `S.true`.

This is the SymPy version of True, for use in the logic module. The primary advantage of using `true` instead of `True` is that shorthand boolean operations like `~` and `>>` will work as expected on this class, whereas with `True` they act bitwise on 1. Functions in the logic module will return this class when they evaluate to true.

See also:

[sympy.logic.boolalg.BooleanFalse](#) (page 720)

Notes

There is liable to be some confusion as to when `True` should be used and when `S.true` should be used in various contexts throughout SymPy. An important thing to remember is that `sympify(True)` returns `S.true`. This means that for the most part, you can just use `True` and it will automatically be converted to `S.true` when necessary, similar to how you can generally use 1 instead of `S.One`.

The rule of thumb is:

“If the boolean in question can be replaced by an arbitrary symbolic Boolean, like `Or(x, y)` or `x > 1`, use `S.true`. Otherwise, use `True`”

In other words, use `S.true` only on those contexts where the boolean is being used as a symbolic representation of truth. For example, if the object ends up in the `.args` of any expression, then it must necessarily be `S.true` instead of `True`, as elements of `.args` must be `Basic`. On the other hand, `==` is not a symbolic operation in SymPy, since it always returns `True` or `False`, and does so in terms of structural equality rather than mathematical, so it should return `True`. The assumptions system should use `True` and `False`. Aside from not satisfying the above rule of thumb, the assumptions system uses a three-valued logic (`True`, `False`, `None`), whereas `S.true` and `S.false` represent a two-valued logic. When in doubt, use `True`.

“`S.true == True` is `True`.”

While “`S.true is True`” is False, “`S.true == True`” is True, so if there is any doubt over whether a function or expression will return `S.true` or `True`, just use `==` instead of `is` to do the comparison, and it will work in either case. Finally, for boolean flags, it's better to just use `if x` instead of `if x is True`. To quote PEP 8:

Don't compare boolean values to `True` or `False` using `==`.

- Yes: `if greeting:`
- No: `if greeting == True:`
- Worse: `if greeting is True:`

Examples

```
>>> from sympy import sympify, true, false, Or
>>> sympify(True)
True
>>> _ is True, _ is true
(False, True)
```

```
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for `true` but a bitwise result for `True`

```
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

Python operators give a boolean result for `true` but a bitwise result for `True`

```
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

class `sympy.logic.boolalg.BooleanFalse`

SymPy version of `False`, a singleton that can be accessed via `S.false`.

This is the SymPy version of `False`, for use in the logic module. The primary advantage of using `false` instead of `False` is that shorthand boolean operations like `~` and `>>` will work as expected on this class, whereas with `False` they act bitwise on 0. Functions in the logic module will return this class when they evaluate to false.

See also:

[sympy.logic.boolalg.BooleanTrue](#) (page 719)

Notes

See note in `:py:class'sympy.logic.boolalg.BooleanTrue'`

Examples

```
>>> from sympy import sympify, true, false, Or
>>> sympify(False)
False
>>> _ is False, _ is false
(False, True)
```

```
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for false but a bitwise result for False

```
>>> ~false, ~False
(True, -1)
>>> false >> false, False >> False
(True, 0)
```

class sympy.logic.boolalg.And
Logical AND function.

It evaluates its arguments in order, giving False immediately if any of them are False, and True if they are all True.

Notes

The & operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, And(a, b) and a & b will return different things if a and b are integers.

```
>>> And(x, y).subs(x, 1)
y
```

Examples

```
>>> from sympy.core import symbols
>>> from sympy.abc import x, y
>>> from sympy.logic.boolalg import And
>>> x & y
x & y
```

class sympy.logic.boolalg.Or
Logical OR function

It evaluates its arguments in order, giving True immediately if any of them are True, and False if they are all False.

Notes

The | operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, Or(a, b) and a | b will return

different things if a and b are integers.

```
>>> Or(x, y).subs(x, 0)
y
```

Examples

```
>>> from sympy.core import symbols
>>> from sympy.abc import x, y
>>> from sympy.logic.boolalg import Or
>>> x | y
x | y
```

class sympy.logic.boolalg.Not

Logical Not function (negation)

Returns True if the statement is False Returns False if the statement is True

Notes

- The $\sim$ operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise not. In particular, $\sim a$ and $\text{Not}(a)$ will be different if a is an integer. Furthermore, since bools in Python subclass from int, $\sim \text{True}$ is the same as ~ 1 which is -2 , which has a boolean value of True. To avoid this issue, use the SymPy boolean types true and false.

```
>>> from sympy import true
>>> ~True
-2
>>> ~true
False
```

Examples

```
>>> from sympy.logic.boolalg import Not, And, Or
>>> from sympy.abc import x, A, B
>>> Not(True)
False
>>> Not(False)
True
>>> Not(And(True, False))
True
>>> Not(Or(True, False))
False
>>> Not(And(And(True, x), Or(x, False)))
~x
>>> ~x
~x
>>> Not(And(Or(A, B), Or(~A, ~B)))
~((A | B) & (~A | ~B))
```


class sympy.logic.boolalg.Xor

Logical XOR (exclusive OR) function.

Returns True if an odd number of the arguments are True and the rest are False.

Returns False if an even number of the arguments are True and the rest are False.

Notes

The $\wedge$ operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, $a \wedge b$ and `Xor(a, b)` will be different if a and b are integers.

```
>>> Xor(x, y).subs(y, 0)
x
```

Examples

```
>>> from sympy.logic.boolalg import Xor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Xor(True, False)
True
>>> Xor(True, True)
False
>>> Xor(True, False, True, True, False)
True
>>> Xor(True, False, True, False)
False
>>> x ^ y
Xor(x, y)
```

class sympy.logic.boolalg.Nand

Logical NAND function.

It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False Returns False if all arguments are True

Examples

```
>>> from sympy.logic.boolalg import Nand
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Nand(False, True)
True
>>> Nand(True, True)
False
>>> Nand(x, y)
~(x & y)
```

class sympy.logic.boolalg.Nor

Logical NOR function.

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True Returns True if all arguments are False

Examples

```
>>> from sympy.logic.boolalg import Nor
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

```
>>> Nor(True, False)
False
>>> Nor(True, True)
False
>>> Nor(False, True)
False
>>> Nor(False, False)
True
>>> Nor(x, y)
~(x | y)
```

class sympy.logic.boolalg.Implies

Logical implication.

A implies B is equivalent to $\neg A \vee B$

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False Returns True otherwise.

Notes

The `>>` and `<<` operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, `Implies(a, b)` and `a >> b` will return different things if `a` and `b` are integers. In particular, since Python considers True and False to be integers, `True >> True` will be the same as `1 >> 1`, i.e., 0, which has a truth value of False. To avoid this issue, use the SymPy objects `true` and `false`.

```
>>> from sympy import true, false
>>> True >> False
1
>>> true >> false
False
```

Examples

```
>>> from sympy.logic.boolalg import Implies
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

```

>>> Implies(True, False)
False
>>> Implies(False, False)
True
>>> Implies(True, True)
True
>>> Implies(False, True)
True
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)

```

class sympy.logic.boolalg.Equivalent

Equivalence relation.

Equivalent(A, B) is True iff A and B are both True or both False

Returns True if all of the arguments are logically equivalent. Returns False otherwise.

Examples

```

>>> from sympy.logic.boolalg import Equivalent, And
>>> from sympy.abc import x, y
>>> Equivalent(False, False, False)
True
>>> Equivalent(True, False, False)
False
>>> Equivalent(x, And(x, True))
True

```

class sympy.logic.boolalg.ITE

If then else clause.

ITE(A, B, C) evaluates and returns the result of B if A is true else it returns the result of C. All args must be Booleans.

Examples

```

>>> from sympy.logic.boolalg import ITE, And, Xor, Or
>>> from sympy.abc import x, y, z
>>> ITE(True, False, True)
False
>>> ITE(Or(True, False), And(True, True), Xor(True, True))
True
>>> ITE(x, y, z)
ITE(x, y, z)
>>> ITE(True, x, y)
x
>>> ITE(False, x, y)
y
>>> ITE(x, y, y)
y

```

Trying to use non-Boolean args will generate a TypeError:

```
>>> ITE(True, [], ())
Traceback (most recent call last):
...
TypeError: expecting bool, Boolean or ITE, not '[]'
```

The following functions can be used to handle Conjunctive and Disjunctive Normal forms-

`sympy.logic.boolalg.to_cnf(expr, simplify=False)`

Convert a propositional logical sentence s to conjunctive normal form. That is, of the form $((A \mid \sim B \mid \dots) \& (B \mid C \mid \dots) \& \dots)$ If `simplify` is `True`, the `expr` is evaluated to its simplest CNF form.

Examples

```
>>> from sympy.logic.boolalg import to_cnf
>>> from sympy.abc import A, B, D
>>> to_cnf(~(A | B) | D)
(D | ~A) & (D | ~B)
>>> to_cnf((A | B) & (A | ~A), True)
A | B
```

`sympy.logic.boolalg.to_dnf(expr, simplify=False)`

Convert a propositional logical sentence s to disjunctive normal form. That is, of the form $((A \& \sim B \& \dots) \mid (B \& C \& \dots) \mid \dots)$ If `simplify` is `True`, the `expr` is evaluated to its simplest DNF form.

Examples

```
>>> from sympy.logic.boolalg import to_dnf
>>> from sympy.abc import A, B, C
>>> to_dnf(B & (A | C))
(A & B) | (B & C)
>>> to_dnf((A & B) | (A & ~B) | (B & C) | (~B & C), True)
A | C
```

`sympy.logic.boolalg.is_cnf(expr)`

Test whether or not an expression is in conjunctive normal form.

Examples

```
>>> from sympy.logic.boolalg import is_cnf
>>> from sympy.abc import A, B, C
>>> is_cnf(A | B | C)
True
>>> is_cnf(A & B & C)
True
>>> is_cnf((A & B) | C)
False
```

`sympy.logic.boolalg.is_dnf(expr)`

Test whether or not an expression is in disjunctive normal form.

Examples

```
>>> from sympy.logic.boolalg import is_dnf
>>> from sympy.abc import A, B, C
>>> is_dnf(A | B | C)
True
>>> is_dnf(A & B & C)
True
>>> is_dnf((A & B) | C)
True
>>> is_dnf(A & (B | C))
False
```

5.16.4 Simplification and equivalence-testing

`sympy.logic.boolalg.simplify_logic(expr, form=None, deep=True)`

This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an Or or And object in SymPy.

Parameters **expr** : string or boolean expression

form : string ('cnf' or 'dnf') or None (default).

If 'cnf' or 'dnf', the simplest expression in the corresponding normal form is returned; if None, the answer is returned according to the form with fewest args (in CNF by default).

deep : boolean (default True)

indicates whether to recursively simplify any non-boolean functions contained within the input.

Examples

```
>>> from sympy.logic import simplify_logic
>>> from sympy.abc import x, y, z
>>> from sympy import S
>>> b = (~x & ~y & ~z) | (~x & ~y & z)
>>> simplify_logic(b)
~x & ~y
```

```
>>> S(b)
(z & ~x & ~y) | (~x & ~y & ~z)
>>> simplify_logic(_)
~x & ~y
```

SymPy's `simplify()` function can also be used to simplify logic expressions to their simplest forms.

`sympy.logic.boolalg.bool_map(bool1, bool2)`

Return the simplified version of `bool1`, and the mapping of variables that makes the two expressions `bool1` and `bool2` represent the same logical behaviour for some correspondence between the variables of each. If more than one mappings of this sort exist, one of them is returned. For example, `And(x, y)` is logically equivalent to `And(a, b)` for the mapping `{x: a, y:b}` or `{x: b, y:a}`. If no such mapping exists, return `False`.

Examples

```
>>> from sympy import SOPform, bool_map, Or, And, Not, Xor
>>> from sympy.abc import w, x, y, z, a, b, c, d
>>> function1 = SOPform([x, z, y], [[1, 0, 1], [0, 0, 1]])
>>> function2 = SOPform([a, b, c], [[1, 0, 1], [1, 0, 0]])
>>> bool_map(function1, function2)
(y & ~z, {y: a, z: b})
```

The results are not necessarily unique, but they are canonical. Here, (w, z) could be (a, d) or (d, a) :

```
>>> eq = Or(And(Not(y), w), And(Not(y), z), And(x, y))
>>> eq2 = Or(And(Not(c), a), And(Not(c), d), And(b, c))
>>> bool_map(eq, eq2)
((x & y) | (w & ~y) | (z & ~y), {w: a, x: b, y: c, z: d})
>>> eq = And(Xor(a, b), c, And(c, d))
>>> bool_map(eq, eq.subs(c, x))
(c & d & (a | b) & (~a | ~b), {a: a, b: b, c: d, d: x})
```

5.16.5 Inference

This module implements some inference routines in propositional logic.

The function `satisfiable` will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence *True*.

For example, the expression $x \ \& \ \sim x$ is not satisfiable, since there are no values for x that make this sentence *True*. On the other hand, $(x \mid y) \ \& \ (x \mid \sim y) \ \& \ (\sim x \mid y)$ is satisfiable with both x and y being *True*.

```
>>> from sympy.logic.inference import satisfiable
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> satisfiable(x & ~x)
False
>>> satisfiable((x | y) & (x | ~y) & (~x | y))
{x: True, y: True}
```

As you see, when a sentence is satisfiable, it returns a model that makes that sentence *True*. If it is not satisfiable it will return *False*.

`sympy.logic.inference.satisfiable(expr, algorithm='dpll2', all_models=False)`

Check satisfiability of a propositional sentence. Returns a model when it succeeds. Returns `{true: true}` for trivially true expressions.

On setting `all_models` to *True*, if given `expr` is satisfiable then returns a generator of models. However, if `expr` is unsatisfiable then returns a generator containing the single element *False*.

Examples

```

>>> from sympy.abc import A, B
>>> from sympy.logic.inference import satisfiable
>>> satisfiable(A & ~B)
{A: True, B: False}
>>> satisfiable(A & ~A)
False
>>> satisfiable(True)
{True: True}
>>> next(satisfiable(A & ~A, all_models=True))
False
>>> models = satisfiable((A >> B) & B, all_models=True)
>>> next(models)
{A: False, B: True}
>>> next(models)
{A: True, B: True}
>>> def use_models(models):
...     for model in models:
...         if model:
...             # Do something with the model.
...             print(model)
...         else:
...             # Given expr is unsatisfiable.
...             print("UNSAT")
>>> use_models(satisfiable(A >> ~A, all_models=True))
{A: False}
>>> use_models(satisfiable(A ^ A, all_models=True))
UNSAT

```

5.17 Matrices

A module that handles matrices.

Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

Contents:

5.17.1 Matrices (linear algebra)

Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first Matrix object:

```

>>> from sympy.interactive.printing import init_printing
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
>>> M = Matrix([[1,0,0], [0,0,0]]); M
[1  0  0]
[  0  0]
[0  0  0]
>>> Matrix([M, (0, 0, -1)])
[1  0  0 ]
[  0  0 ]

```

(continues on next page)

(continued from previous page)

```
[0 0 0 ]
[      ]
[0 0 -1]
>>> Matrix([[1, 2, 3]])
[1 2 3]
>>> Matrix([1, 2, 3])
[1]
[ ]
[2]
[ ]
[3]
```

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, SymPy also supports more advanced methods of matrix creation including a single list of values and dimension inputs:

```
>>> Matrix(2, 3, [1, 2, 3, 4, 5, 6])
[1 2 3]
[      ]
[4 5 6]
```

More interesting (and useful), is the ability to use a 2-variable function (or `lambda`) to create a matrix. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```
>>> def f(i,j):
...     if i == j:
...         return 1
...     else:
...         return 0
...
>>> Matrix(4, 4, f)
[1 0 0 0]
[      ]
[0 1 0 0]
[      ]
[0 0 1 0]
[      ]
[0 0 0 1]
```

Finally let's use `lambda` to create a 1-line matrix with 1's in the even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)
[1 0 1 0]
[      ]
[0 1 0 1]
[      ]
[1 0 1 0]
```

There are also a couple of special constructors for quick matrix construction: `eye` is the identity matrix, `zeros` and `ones` for matrices of all zeros and ones, respectively, and `diag` to put matrices or elements along the diagonal:

```
>>> eye(4)
[1 0 0 0]
[      ]
```

(continues on next page)

(continued from previous page)

```

[0  1  0  0]
[
  ]
[0  0  1  0]
[
  ]
[0  0  0  1]
>>> zeros(2)
[0  0]
[
  ]
[0  0]
>>> zeros(2, 5)
[0  0  0  0  0]
[
  ]
[0  0  0  0  0]
>>> ones(3)
[1  1  1]
[
  ]
[1  1  1]
[
  ]
[1  1  1]
>>> ones(1, 3)
[1  1  1]
>>> diag(1, Matrix([[1, 2], [3, 4]]))
[1  0  0]
[
  ]
[0  1  2]
[
  ]
[0  3  4]

```

Basic Manipulation

While learning to work with matrices, let's choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

```

>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5

```

Now, the more standard entry access is a pair of indices which will always return the value at the corresponding row and column of the matrix:

```

>>> M[1, 2]
6
>>> M[0, 0]
1
>>> M[1, 1]
5

```

Since this is Python we're also able to slice submatrices; slices always give a matrix in return, even if the dimension is 1 x 1:

```

>>> M[0:2, 0:2]
[1  2]

```

(continues on next page)

(continued from previous page)

```
[
]
[4 5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[ ]
[6]
>>> M[:1, 2]
[3]
```

In the second example above notice that the slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) the first row/column is 0.

You cannot access rows or columns that are not present unless they are in a slice:

```
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...
IndexError: Index out of range: a[[0, 10]]
>>> M[:, 10:11] # the 10-th column (if there)
[]
>>> M[:, :10] # all columns up to the 10-th
[1 2 3]
[ ]
[4 5 6]
```

Slicing an empty matrix works as long as you use a slice for the coordinate that has no size:

```
>>> Matrix(0, 3, [])[:, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of one object do not affect the other:

```
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice that changing M2 didn't change M. Since we can slice, we can also assign entries:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1 2 3 4]
[ ]
[5 6 7 8]
[ ]
[9 10 11 12]
[ ]
[13 14 15 16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1 2 3 0]
[ ]
[5 6 7 8]
[ ]
[9 10 0 12]
```

(continues on next page)

(continued from previous page)

```
[
[13 14 15 16]
```

as well as assign slices:

```
>>> M = Matrix([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
>>> M[2:,2:] = Matrix(2,2,lambdai,j: 0)
>>> M
[1  2  3  4]
[
[5  6  7  8]
[
[9  10 0  0]
[
[13 14 0  0]
```

All the standard arithmetic operations are supported:

```
>>> M = Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> M - M
[0 0 0]
[
[0 0 0]
[
[0 0 0]
>>> M + M
[2  4  6 ]
[
[8  10 12]
[
[14 16 18]
>>> M * M
[30  36  42 ]
[
[66  81  96 ]
[
[102 126 150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[ ]
[29]
[ ]
[47]
>>> M**2
[30  36  42 ]
[
[66  81  96 ]
[
[102 126 150]
```

As well as some useful vector operations:

```
>>> M.row_del(0)
>>> M
[4  5  6]
[
```

(continues on next page)

(continued from previous page)

```
[7  8  9]
>>> M.col_del(1)
>>> M
[4  6]
[   ]
[7  9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0
```

Recall that the `row_del()` and `col_del()` operations don't return a value - they simply change the matrix object. We can also “glue” together matrices of the appropriate size:

```
>>> M1 = eye(3)
>>> M2 = zeros(3, 4)
>>> M1.row_join(M2)
[1  0  0  0  0  0  0]
[   ]
[0  1  0  0  0  0  0]
[   ]
[0  0  1  0  0  0  0]
>>> M3 = zeros(4, 3)
>>> M1.col_join(M3)
[1  0  0]
[   ]
[0  1  0]
[   ]
[0  0  1]
[   ]
[0  0  0]
[   ]
[0  0  0]
[   ]
[0  0  0]
[   ]
[0  0  0]
[   ]
[0  0  0]
```

Operations on entries

We are not restricted to having multiplication between two matrices:

```
>>> M = eye(3)
>>> 2*M
[2  0  0]
[   ]
[0  2  0]
[   ]
[0  0  2]
>>> 3*M
```

(continues on next page)

(continued from previous page)

```
[3  0  0]
[   ]
[0  3  0]
[   ]
[0  0  3]
```

but we can also apply functions to our matrix entries using `applyfunc()`. Here we'll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```
>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2  0  0]
[   ]
[0  2  0]
[   ]
[0  0  2]
```

One more useful matrix-wide entry application function is the substitution function. Let's declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!:

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x  0  0]
[   ]
[0  x  0]
[   ]
[0  0  x]
>>> M.subs(x, 4)
[4  0  0]
[   ]
[0  4  0]
[   ]
[0  0  4]
>>> y = Symbol('y')
>>> M.subs(x, y)
[y  0  0]
[   ]
[0  y  0]
[   ]
[0  0  y]
```

Linear algebra

Now that we have the basics out of the way, let's see what we can do with the actual matrices. Of course, one of the first things that comes to mind is the determinant:

```
>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
```

(continues on next page)

(continued from previous page)

```

1
>>> M3 = Matrix(( [1, 0, 0], [1, 0, 0], [1, 0, 0] ))
>>> M3.det()
0

```

Another common operation is the inverse: In SymPy, this is computed by Gaussian elimination by default (for dense matrices) but we can specify it be done by *LU* decomposition as well:

```

>>> M2.inv()
[1  0  0]
[  0  1  0]
[0  0  1]
>>> M2.inv(method="LU")
[1  0  0]
[  0  1  0]
[0  0  1]
>>> M.inv(method="LU")
[-3/14  1/14  1/2 ]
[  0  0  0]
[-1/28  5/28 -1/4]
[  0  0  0]
[ 3/7  -1/7  0 ]
>>> M * M.inv(method="LU")
[1  0  0]
[  0  1  0]
[0  0  1]

```

We can perform a *QR* factorization which is handy for solving systems:

```

>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[
[ $\sqrt{6}$    $-\sqrt{3}$    $-\sqrt{2}$  ]
[-----]
[  6      3      2 ]
[
[ $\sqrt{6}$    $-\sqrt{3}$    $\sqrt{2}$  ]
[-----]
[  6      3      2 ]
[
[ $\sqrt{6}$    $\sqrt{3}$   0 ]
[-----]
[  3      3 ]
]
>>> R
[
[  0   $4*\sqrt{6}$  ]
[ $\sqrt{6}$   -----   $2*\sqrt{6}$  ]
[  3      3 ]
]

```

(continues on next page)

(continued from previous page)

```

[
[
[
[ 0       $\frac{\sqrt{3}}{3}$       0 ]
[
[
[ 0      0       $\sqrt{2}$  ]
>>> Q*R
[1  1  1]
[
[1  1  3]
[
[2  3  4]

```

In addition to the solvers in the `solver.py` file, we can solve the system $Ax=b$ by passing the `b` vector to the matrix `A`'s `LUsolve` function. Here we'll cheat a little choose `A` and `x` then multiply to get `b`. Then we can solve for `x` and check that it's correct:

```

>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]

```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to `False`. Let's take some vectors and orthogonalize them - one normalized and one not:

```

>>> L = [Matrix([2,3,5]), Matrix([3,6,2]), Matrix([8,3,6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)

```

Let's take a look at the vectors:

```

>>> for i in out1:
...     print(i)
...
Matrix([[2], [3], [5]])
Matrix([[23/19], [63/19], [-47/19]])
Matrix([[1692/353], [-1551/706], [-423/706]])
>>> for i in out2:
...     print(i)
...
Matrix([[sqrt(38)/19], [3*sqrt(38)/38], [5*sqrt(38)/38]])
Matrix([[23*sqrt(6707)/6707], [63*sqrt(6707)/6707], [-47*sqrt(6707)/6707]])
Matrix([[12*sqrt(706)/353], [-11*sqrt(706)/706], [-3*sqrt(706)/706]])

```

We can spot-check their orthogonality with `dot()` and their normality with `norm()`:

```
>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1
```

So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the `matrices.py` file for all functionality.

MatrixBase Class Reference

class `sympy.matrices.matrices.MatrixBase`
Base class for matrix objects.

Attributes

<code>cols</code>	
<code>rows</code>	

D

Return Dirac conjugate (if `self.rows == 4`).

See also:

conjugate By-element conjugation

H Hermite conjugation

Examples

```
>>> from sympy import Matrix, I, eye
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m.D
Matrix([[0, 1 - I, -2, -3]])
>>> m = (eye(4) + I*eye(4))
>>> m[0, 3] = 2
>>> m.D
Matrix([
[1 - I,      0,      0,      0],
[      0, 1 - I,      0,      0],
[      0,      0, -1 + I,      0],
[      2,      0,      0, -1 + I]])
```

If the matrix does not have 4 rows an `AttributeError` will be raised because this property is only defined for matrices with 4 rows.


```
>>> Matrix(eye(2)).D
Traceback (most recent call last):
...
AttributeError: Matrix has no attribute D.
```

LDLdecomposition()

Returns the LDL Decomposition (L, D) of matrix A, such that $L * D * L.H == A$. This method eliminates the use of square root. Further this ensures that all the diagonal entries of L are 1. A must be a Hermitian positive-definite matrix.

See also:

[cholesky](#) (page 743), [LUdecomposition](#) (page 740), [QRdecomposition](#) (page 742)

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1, 0, 0],
[ 3/5, 1, 0],
[-1/5, 1/3, 1]])
>>> D
Matrix([
[25, 0, 0],
[ 0, 9, 0],
[ 0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1, 0],
[-I/3, 1]])
>>> D
Matrix([
[9, 0],
[0, 4]])
>>> L*D*L.H == A
True
```

LDLsolve(rhs)

Solves $Ax = B$ using LDL decomposition, for a general square and non-singular matrix.

For a non-square matrix with rows > cols, the least squares solution is returned.

See also:

[LDLdecomposition](#) (page 739), [lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745),

[cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LUsolve](#) (page 741), [QRsolve](#) (page 742), [pinv_solve](#) (page 750)

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True
```

LUdecomposition(iszerofunc=<function `_iszero`>, simpfunc=None, rankcheck=False)

Returns (L, U, perm) where L is a lower triangular matrix with unit diagonal, U is an upper triangular matrix, and perm is a list of row swap index pairs. If A is the original matrix, then $A = (L*U).permuteBkwd(perm)$, and the row permutation matrix P such that $P*A = L*U$ can be computed by $P=eye(A.row).permuteFwd(perm)$.

See documentation for LUCombined for details about the keyword argument rankcheck, iszerofunc, and simpfunc.

See also:

[cholesky](#) (page 743), [LDLdecomposition](#) (page 739), [QRdecomposition](#) (page 742), [LUdecomposition_Simple](#) (page 741), [LUdecompositionFF](#) (page 740), [LUsolve](#) (page 741)

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[4, 3], [6, 3]])
>>> L, U, _ = a.LUdecomposition()
>>> L
Matrix([
[ 1, 0],
[3/2, 1]])
>>> U
Matrix([
[4, 3],
[0, -3/2]])
```

LUdecompositionFF()

Compute a fraction-free LU decomposition.

Returns 4 matrices P, L, D, U such that $PA = L D^{-1} U$. If the elements of the matrix belong to some integral domain I, then all elements of L, D and U are guaranteed to belong to I.

Reference

- W. Zhou & D.J. Jeffrey, "Fraction-free matrix factors: new forms for LU and QR factors". Frontiers in Computer Science in China, Vol 2, no. 1, pp. 67-80, 2008.

See also:

[LUdecomposition](#) (page 740), [LUdecomposition_Simple](#) (page 741), [LUsolve](#) (page 741)

LUdecomposition_Simple(iszerofunc=<function _iszero>, simpfunc=None, rankcheck=False)

Compute an lu decomposition of $m \times n$ matrix A , where $P*A = L*U$

- L is $m \times m$ lower triangular with unit diagonal
- U is $m \times n$ upper triangular
- P is an $m \times m$ permutation matrix

Returns an $m \times n$ matrix lu , and an m element list $perm$ where each element of $perm$ is a pair of row exchange indices.

The factors L and U are stored in lu as follows: The subdiagonal elements of L are stored in the subdiagonal elements of lu , that is $lu[i, j] = L[i, j]$ whenever $i > j$. The elements on the diagonal of L are all 1, and are not explicitly stored. U is stored in the upper triangular portion of lu , that is $lu[i, j] = U[i, j]$ whenever $i \leq j$. The output matrix can be visualized as:

Matrix([[u, u, u, u], [l, u, u, u], [l, l, u, u], [l, l, l, u]])

where l represents a subdiagonal entry of the L factor, and u represents an entry from the upper triangular entry of the U factor.

$perm$ is a list row swap index pairs such that if A is the original matrix, then $A = (L*U).permuteBkwd(perm)$, and the row permutation matrix P such that $P*A = L*U$ can be computed by $P=eye(A.row).permuteFwd(perm)$.

The keyword argument `rankcheck` determines if this function raises a `ValueError` when passed a matrix whose rank is strictly less than `min(num rows, num cols)`. The default behavior is to decompose a rank deficient matrix. Pass `rankcheck=True` to raise a `ValueError` instead. (This mimics the previous behavior of this function).

The keyword arguments `iszerofunc` and `simpfunc` are used by the pivot search algorithm. `iszerofunc` is a callable that returns a boolean indicating if its input is zero, or `None` if it cannot make the determination. `simpfunc` is a callable that simplifies its input. The default is `simpfunc=None`, which indicate that the pivot search algorithm should not attempt to simplify any candidate pivots. If `simpfunc` fails to simplify its input, then it must return its input instead of a copy.

When a matrix contains symbolic entries, the pivot search algorithm differs from the case where every entry can be categorized as zero or nonzero. The algorithm searches column by column through the submatrix whose top left entry coincides with the pivot position. If it exists, the pivot is the first entry in the current search column that `iszerofunc` guarantees is nonzero. If no such candidate exists, then each candidate pivot is simplified if `simpfunc` is not `None`. The search is repeated, with the difference that a candidate may be the pivot if `iszerofunc()` cannot guarantee that it is nonzero. In the second search the pivot is the first candidate that `iszerofunc` can guarantee is nonzero. If no such candidate exists, then the pivot is the first candidate for which `iszerofunc` returns `None`. If no such candidate exists, then the search is repeated in the next column to the right. The pivot search algorithm differs from the one in `rref()`, which relies on `_find_reasonable_pivot()`. Future versions of `LUdecomposition_simple()` may use `_find_reasonable_pivot()`.

See also:

[LUdecomposition](#) (page 740), [LUdecompositionFF](#) (page 740), [LUsolve](#) (page 741)

LUsolve(rhs, iszerofunc=<function _iszero>)

Solve the linear system $Ax = rhs$ for x where $A = self$.

This is for symbolic matrices, for real or complex ones use `mpmath.lu_solve` or `mpmath.qr_solve`.

See also:

[lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745), [cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739), [QRsolve](#) (page 742), [pinv_solve](#) (page 750), [LUdecomposition](#) (page 740)

QRdecomposition()

Return Q, R where $A = Q \cdot R$, Q is orthogonal and R is upper triangular.

See also:

[cholesky](#) (page 743), [LDLdecomposition](#) (page 739), [LUdecomposition](#) (page 740), [QRsolve](#) (page 742)

Examples

This is the example from wikipedia:

```
>>> from sympy import Matrix
>>> A = Matrix([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[ 6/7, -69/175, -58/175],
[ 3/7, 158/175,  6/175],
[-2/7,  6/35, -33/35]])
>>> R
Matrix([
[14, 21, -14],
[ 0, 175, -70],
[ 0,  0, 35]])
>>> A == Q*R
True
```

QR factorization of an identity matrix:

```
>>> A = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> R
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

QRsolve(b)

Solve the linear system $Ax = b$.

'self' is the matrix 'A', the method argument is the vector 'b'. The method returns the solution vector 'x'. If 'b' is a matrix, the system is solved for each column of 'b' and the return value is a matrix of the same shape as 'b'.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact

arithmetic, so you don't need to use `QRsolve`.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use `mpmath.qr_solve`.

See also:

[lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745), [cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739), [LUsolve](#) (page 741), [pinv_solve](#) (page 750), [QRdecomposition](#) (page 742)

add(b)

Return self + b

cholesky()

Returns the Cholesky decomposition L of a matrix A such that $L * L.H = A$

A must be a Hermitian positive-definite matrix.

See also:

[LDLdecomposition](#) (page 739), [LUdecomposition](#) (page 740), [QRdecomposition](#) (page 742)

Examples

```
>>> from sympy.matrices import Matrix
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T
Matrix([
[25, 15, -5],
[15, 18, 0],
[-5, 0, 11]])
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> A.cholesky()
Matrix([
[ 3, 0],
[-I, 2]])
>>> A.cholesky() * A.cholesky().H
Matrix([
[ 9, 3*I],
[-3*I, 5]])
```

cholesky_solve(rhs)

Solves $Ax = B$ using Cholesky decomposition, for a general square non-singular matrix. For a non-square matrix with rows > cols, the least squares solution is returned.

See also:

[lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739),

[LUsolve](#) (page 741), [QRsolve](#) (page 742), [pinv_solve](#) (page 750)

condition_number()

Returns the condition number of a matrix.

This is the maximum singular value divided by the minimum singular value

See also:

[singular_values](#)

Examples

```
>>> from sympy import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

copy()

Returns the copy of a matrix.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.copy()
Matrix([
[1, 2],
[3, 4]])
```

cross(b)

Return the cross product of `self` and `b` relaxing the condition of compatible dimensions: if each has 3 elements, a matrix of the same type and shape as `self` will be returned. If `b` has the same shape as `self` then common identities for the cross product (like $a \times b = -b \times a$) will hold.

Parameters `b`: 3x1 or 1x3 Matrix

See also:

[dot](#) (page 745), [multiply](#) (page 748), [multiply_elementwise](#)

diagonal_solve(rhs)

Solves $Ax = B$ efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

See also:

[lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745), [cholesky_solve](#) (page 743), [LDLsolve](#) (page 739), [LUsolve](#) (page 741), [QRsolve](#) (page 742), [pinv_solve](#) (page 750)

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True
```

dot(b)

Return the dot product of two vectors of equal length. `self` must be a `Matrix` of size $1 \times n$ or $n \times 1$, and `b` must be either a matrix of size $1 \times n$, $n \times 1$, or a list/tuple of length n . A scalar is returned.

See also:

`cross` (page 744), `multiply` (page 748), `multiply_elementwise`

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = Matrix([1, 1, 1])
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> v = [3, 2, 1]
>>> M.row(0).dot(v)
10
```

dual()

Returns the dual of a matrix, which is:

$(1/2) * \text{levicivita}(i, j, k, l) * M(k, l)$ summed over indices k and l

Since the `levicivita` method is `anti_symmetric` for any pairwise exchange of indices, the dual of a symmetric matrix is the zero matrix. Strictly speaking the dual defined here assumes that the ‘matrix’ M is a contravariant `anti_symmetric` second rank tensor, so that the dual is a covariant second rank tensor.

exp()

Return the exponentiation of a square matrix.

gauss_jordan_solve(b, freevar=False)

Solves $Ax = b$ using Gauss Jordan elimination.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, it will be returned parametrically. If no solutions exist, It will throw `ValueError`.

Parameters **b** : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A .

freevar : List

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary values of free variables. Then the index of the free variables in the solutions (column Matrix) will be returned by `freevar`, if the flag `freevar` is set to `True`.

Returns **x** : Matrix

The matrix that will satisfy $Ax = B$. Will have as many rows as matrix A has columns, and as many columns as matrix B.

params : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary parameters. These arbitrary parameters are returned as params Matrix.

See also:

[lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739), [LU_solve](#) (page 741), [QRsolve](#) (page 742), [pinv](#) (page 749)

References

[R391] (page 1865)

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> b = Matrix([7, 12, 4])
>>> sol, params = A.gauss_jordan_solve(b)
>>> sol
Matrix([
[-2*tau0 - 3*tau1 + 2],
[      tau0],
[      2*tau1 + 5],
[      tau1]])
>>> params
Matrix([
[tau0],
[tau1]])
```

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> sol, params = A.gauss_jordan_solve(b)
>>> sol
Matrix([
[-1],
[ 2],
[ 0]])
>>> params
Matrix(0, 1, [])
```

inv(method=None, **kwargs)

Return the inverse of a matrix.

CASE 1: If the matrix is a dense matrix.

Return the matrix inverse using the method indicated (default is Gauss elimination).

Parameters **method** : ('GE', 'LU', or 'ADJ')

Raises ValueError

If the determinant of the matrix is zero.

See also:

[inverse_LU](#) (page 747), [inverse_GE](#) (page 747), [inverse_ADJ](#) (page 747)

Notes

According to the method keyword, it calls the appropriate method:

LDL ... `inverse_LDL()`; default CH `inverse_CH()`

Kwargs

method : ('CH', 'LDL')

inv_mod(m)

Returns the inverse of the matrix K (mod m), if it exists.

Method to find the matrix inverse of K (mod m) implemented in this function:

- Compute $\text{adj}(K) = \text{cof}(K)^t$, the adjoint matrix of K .
- Compute $r = 1/\det(K) \pmod{m}$.
- $K^{-1} = r \cdot \text{adj}(K) \pmod{m}$.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([
[3, 1],
[4, 2]])
>>> A.inv_mod(3)
Matrix([
[1, 1],
[0, 1]])
```

inverse_ADJ(iszerofunc=<function _iszero>)

Calculates the inverse using the adjugate matrix and a determinant.

See also:

[inv](#) (page 746), [inverse_LU](#) (page 747), [inverse_GE](#) (page 747)

inverse_GE(iszerofunc=<function _iszero>)

Calculates the inverse using Gaussian elimination.

See also:

[inv](#) (page 746), [inverse_LU](#) (page 747), [inverse_ADJ](#) (page 747)

inverse_LU(iszerofunc=<function _iszero>)

Calculates the inverse using LU decomposition.

See also:

[inv](#) (page 746), [inverse_GE](#) (page 747), [inverse_ADJ](#) (page 747)

is_nilpotent()

Checks if a matrix is nilpotent.

A matrix B is nilpotent if for some integer k, B^{**k} is a zero matrix.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[0, 0, 0], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
True
```

```
>>> a = Matrix([[1, 0, 1], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
False
```

key2bounds(keys)

Converts a key with potentially mixed types of keys (integer and slice) into a tuple of ranges and raises an error if any index is out of self's range.

See also:

[key2ij](#) (page 748)

key2ij(key)

Converts key into canonical form, converting integers or indexable items into valid integers for self's range or returning slices unchanged.

See also:

[key2bounds](#) (page 748)

lower_triangular_solve(rhs)

Solves $Ax = B$, where A is a lower triangular matrix.

See also:

[upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745), [cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739), [LUsolve](#) (page 741), [QRsolve](#) (page 742), [pinv_solve](#) (page 750)

multiply(b)

Returns $\text{self} * b$

See also:

[dot](#) (page 745), [cross](#) (page 744), [multiply_elementwise](#)

norm(ord=None)

Return the Norm of a Matrix or Vector. In the simplest case this is the geometric size of the vector Other norms can be specified by the ord parameter

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	• does not exist
inf	maximum row sum	max(abs(x))
-inf	-	min(abs(x))
1	maximum column sum	as below
-1	-	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	• does not exist	sum(abs(x)**ord)**(1./ord)

See also:

[normalized](#) (page 749)

Examples

```
>>> from sympy import Matrix, Symbol, trigsimp, cos, sin, oo
>>> x = Symbol('x', real=True)
>>> v = Matrix([cos(x), sin(x)])
>>> trigsimp( v.norm() )
1
>>> v.norm(10)
(sin(x)**10 + cos(x)**10)**(1/10)
>>> A = Matrix([[1, 1], [1, 1]])
>>> A.norm(1) # maximum sum of absolute values of A is 2
2
>>> A.norm(2) # Spectral norm (max of |Ax|/|x| under 2-vector-norm)
2
>>> A.norm(-2) # Inverse spectral norm (smallest singular value)
0
>>> A.norm() # Frobenius Norm
2
>>> A.norm(oo) # Infinity Norm
2
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1
```

normalized()

Return the normalized version of self.

See also:

[norm](#) (page 748)

pinv()

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

See also:

[inv](#) (page 746), [pinv_solve](#) (page 750)

References

[R392] (page 1865)

Examples

```
>>> from sympy import Matrix
>>> Matrix([[1, 2, 3], [4, 5, 6]]).pinv()
Matrix([
[-17/18,  4/9],
[ -1/9,  1/9],
[ 13/18, -2/9]])
```

pinv_solve(B, arbitrary_matrix=None)

Solve $Ax = B$ using the Moore-Penrose pseudoinverse.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of `arbitrary_matrix`. If no solutions exist, the least-squares solution is returned.

Parameters **B** : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

arbitrary_matrix : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of `wn_m` will be used, with n and m being row and column position of each symbol.

Returns **x** : Matrix

The matrix that will satisfy $Ax = B$. Will have as many rows as matrix A has columns, and as many columns as matrix B.

See also:

[lower_triangular_solve](#) (page 748), [upper_triangular_solve](#) (page 753), [gauss_jordan_solve](#) (page 745), [cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739), [LUsolve](#) (page 741), [QRsolve](#) (page 742), [pinv](#) (page 749)

Notes

This may return either exact solutions or least squares solutions. To determine which, check `A * A.pinv() * B == B`. It will be True if exact solutions exist, and

False if only a least-squares solution exists. Be aware that the left hand side of that equation may need to be simplified to correctly compare to the right hand side.

References

[R393] (page 1866)

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([
 [ _w0_0/6 - _w1_0/3 + _w2_0/6 - 55/18],
 [-_w0_0/3 + 2*_w1_0/3 - _w2_0/3 + 1/9],
 [ _w0_0/6 - _w1_0/3 + _w2_0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([
 [-55/18],
 [ 1/9],
 [ 59/18]])
```

print_nonzero(symb='X')

Shows location of non-zero entries for fast shape lookup.

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> m = Matrix(2, 3, lambda i, j: i*3+j)
>>> m
Matrix([
 [0, 1, 2],
 [3, 4, 5]])
>>> m.print_nonzero()
[ XX]
[XXX]
>>> m = eye(4)
>>> m.print_nonzero("x")
[x  ]
[ x ]
[  x]
[   x]
```

project(v)

Return the projection of self onto the line containing v.

Examples

```
>>> from sympy import Matrix, S, sqrt
>>> V = Matrix([sqrt(3)/2, S.Half])
>>> x = Matrix([[1, 0]])
>>> V.project(x)
Matrix([[sqrt(3)/2, 0]])
>>> V.project(-x)
Matrix([[sqrt(3)/2, 0]])
```

solve(rhs, method='GE')

Return solution to $\text{self} * \text{soln} = \text{rhs}$ using given inversion method.

For a list of possible inversion methods, see the `.inv()` docstring.

solve_least_squares(rhs, method='CH')

Return the least-square fit to the data.

By default the `cholesky_solve` routine is used (method='CH'); other methods of matrix inversion can be used. To find out which are available, see the docstring of the `.inv()` method.

Examples

```
>>> from sympy.matrices import Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = Matrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of `S` represent coefficients of $Ax + By$ and x and y are `[2, 3]` then $S * xy$ is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of `xy`:

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by $S * xy - r$:

```
>>> S*xy - r
Matrix([
[1/3],
[1/3],
[1/3]])
>>> _.norm().n(2)
0.58
```

If a different `xy` is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

table(printer, rowstart='[', rowend=']', rowsep='\n', colsep=', ', align='right')

String form of Matrix as a table.

`printer` is the printer to use for on the elements (generally something like `StrPrinter()`)

`rowstart` is the string used to start each row (by default '[').

`rowend` is the string used to end each row (by default ']').

`rowsep` is the string used to separate rows (by default a newline).

`colsep` is the string used to separate columns (by default ', ').

`align` defines how the elements are aligned. Must be one of 'left', 'right', or 'center'. You can also use '<', '>', and '^' to mean the same thing, respectively.

This is used by the string printer for Matrix.

Examples

```
>>> from sympy import Matrix
>>> from sympy.printing.str import StrPrinter
>>> M = Matrix([[1, 2], [-33, 4]])
>>> printer = StrPrinter()
>>> M.table(printer)
'[ 1, 2]\n[-33, 4]'
>>> print(M.table(printer))
[ 1, 2]
[-33, 4]
>>> print(M.table(printer, rowsep=',\n'))
[ 1, 2],
[-33, 4]
>>> print('[%s]' % M.table(printer, rowsep=',\n'))
[[ 1, 2],
[-33, 4]]
>>> print(M.table(printer, colsep=' '))
[ 1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1 , 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{ 1, 2}
{-33, 4}
```

upper_triangular_solve(rhs)

Solves $Ax = B$, where A is an upper triangular matrix.

See also:

[lower_triangular_solve](#) (page 748), [gauss_jordan_solve](#) (page 745), [cholesky_solve](#) (page 743), [diagonal_solve](#) (page 744), [LDLsolve](#) (page 739), [LUsolve](#) (page 741), [QRsolve](#) (page 742), [pinv_solve](#) (page 750)

vech(diagonal=True, check_symmetry=True)

Return the unique elements of a symmetric Matrix as a one column matrix by stacking the elements in the lower triangle.

Arguments: diagonal - include the diagonal cells of self or not check_symmetry - checks symmetry of self but not completely reliably

See also:

vec

Examples

```
>>> from sympy import Matrix
>>> m=Matrix([[1, 2], [2, 3]])
>>> m
Matrix([
[1, 2],
[2, 3]])
>>> m.vech()
Matrix([
[1],
[2],
[3]])
>>> m.vech(diagonal=False)
Matrix([[2]])
```

Matrix Exceptions Reference

class sympy.matrices.matrices.**MatrixError**

class sympy.matrices.matrices.**ShapeError**

Wrong matrix shape

class sympy.matrices.matrices.**NonSquareMatrixError**

Matrix Functions Reference

sympy.matrices.matrices.**classof**(A, B)

Get the type of the result when combining matrices of different types.

Currently the strategy is that immutability is contagious.

Examples

```
>>> from sympy import Matrix, ImmutableMatrix
>>> from sympy.matrices.matrices import classof
>>> M = Matrix([[1, 2], [3, 4]]) # a Mutable Matrix
>>> IM = ImmutableMatrix([[1, 2], [3, 4]])
>>> classof(M, IM)
<class 'sympy.matrices.immutable.ImmutableDenseMatrix'>
```

sympy.matrices.dense.**matrix_multiply_elementwise**(A, B)

Return the Hadamard product (elementwise product) of A and B


```
>>> from sympy.matrices import matrix_multiply_elementwise
>>> from sympy.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> matrix_multiply_elementwise(A, B)
Matrix([
[ 0, 10, 200],
[300, 40,  5]])
```

See also:`__mul__``sympy.matrices.dense.zeros(*args, **kwargs)`

Returns a matrix of zeros with rows rows and cols columns; if cols is omitted a square matrix will be returned.

See also:`ones` (page 755), `eye` (page 755), `diag` (page 755)`sympy.matrices.dense.ones(*args, **kwargs)`

Returns a matrix of ones with rows rows and cols columns; if cols is omitted a square matrix will be returned.

See also:`zeros` (page 755), `eye` (page 755), `diag` (page 755)`sympy.matrices.dense.eye(*args, **kwargs)`

Create square identity matrix n x n

See also:`diag` (page 755), `zeros` (page 755), `ones` (page 755)`sympy.matrices.dense.diag(*values, **kwargs)`

Create a sparse, diagonal matrix from a list of diagonal values.

See also:`eye` (page 755)**Notes**

When arguments are matrices they are fitted in resultant matrix.

The returned matrix is a mutable, dense matrix. To make it a different type, send the desired class for keyword `cls`.

Examples

```
>>> from sympy.matrices import diag, Matrix, ones
>>> diag(1, 2, 3)
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> diag(*[1, 2, 3])
Matrix([
```

(continues on next page)

(continued from previous page)

```
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

The diagonal elements can be matrices; diagonal filling will continue on the diagonal from the last element of the matrix:

```
>>> from sympy.abc import x, y, z
>>> a = Matrix([x, y, z])
>>> b = Matrix([[1, 2], [3, 4]])
>>> c = Matrix([[5, 6]])
>>> diag(a, 7, b, c)
Matrix([
[x, 0, 0, 0, 0, 0],
[y, 0, 0, 0, 0, 0],
[z, 0, 0, 0, 0, 0],
[0, 7, 0, 0, 0, 0],
[0, 0, 1, 2, 0, 0],
[0, 0, 3, 4, 0, 0],
[0, 0, 0, 0, 5, 6]])
```

When diagonal elements are lists, they will be treated as arguments to Matrix:

```
>>> diag([1, 2, 3], 4)
Matrix([
[1, 0],
[2, 0],
[3, 0],
[0, 4]])
>>> diag([[1, 2, 3]], 4)
Matrix([
[1, 2, 3, 0],
[0, 0, 0, 4]])
```

A given band off the diagonal can be made by padding with a vertical or horizontal “kerning” vector:

```
>>> hpad = ones(0, 2)
>>> vpad = ones(2, 0)
>>> diag(vpad, 1, 2, 3, hpad) + diag(hpad, 4, 5, 6, vpad)
Matrix([
[0, 0, 4, 0, 0],
[0, 0, 0, 5, 0],
[1, 0, 0, 0, 6],
[0, 2, 0, 0, 0],
[0, 0, 3, 0, 0]])
```

The type is mutable by default but can be made immutable by setting the mutable flag to False:

```
>>> type(diag(1))
<class 'sympy.matrices.dense.MutableDenseMatrix'>
>>> from sympy.matrices import ImmutableMatrix
>>> type(diag(1, cls=ImmutableMatrix))
<class 'sympy.matrices.immutable.ImmutableDenseMatrix'>
```

`sympy.matrices.dense.jordan_cell(eigenval, n)`

Create a Jordan block:

Examples

```
>>> from sympy.matrices import jordan_cell
>>> from sympy.abc import x
>>> jordan_cell(x, 4)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

`sympy.matrices.dense.hessian(f, varlist, constraints=[])`

Compute Hessian matrix for a function f wrt parameters in `varlist` which may be given as a sequence or a row/column vector. A list of constraints may optionally be given.

See also:

`sympy.matrices.mutable.Matrix.jacobian`, [wronskian](#) (page 758)

References

http://en.wikipedia.org/wiki/Hessian_matrix

Examples

```
>>> from sympy import Function, hessian, pprint
>>> from sympy.abc import x, y
>>> f = Function('f')(x, y)
>>> g1 = Function('g')(x, y)
>>> g2 = x**2 + 3*y
>>> pprint(hessian(f, (x, y), [g1, g2]))
[
[
0      0      --(g(x, y))  --(g(x, y))
[
dx      dy
[
0      0      2*x      3
[
2      2
[
d      d
[
--(g(x, y))  2*x  --- (f(x, y))  ---- (f(x, y))
[dx      2      dy dx
[
dx
[
2      2
[
d      d
[
--(g(x, y))  3  ---- (f(x, y))  --- (f(x, y))
[dy      dy dx      2
[
dy
]
```

`sympy.matrices.dense.GramSchmidt(vlist, orthonormal=False)`

Apply the Gram-Schmidt process to a set of vectors.

see: http://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process

`sympy.matrices.dense.wronskian`(functions, var, method='bareiss')

Compute Wronskian for [] of functions

$$W(f_1, \dots, f_n) = \begin{vmatrix} f_1 & f_2 & \dots & f_n \\ f_1' & f_2' & \dots & f_n' \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ D^{(n)}(f_1) & D^{(n)}(f_2) & \dots & D^{(n)}(f_n) \end{vmatrix}$$

see: <http://en.wikipedia.org/wiki/Wronskian>

See also:

`sympy.matrices.mutable.Matrix.jacobian`, [hessian](#) (page 757)

`sympy.matrices.dense.casoratian`(seqs, n, zero=True)

Given linear difference operator L of order ' k ' and homogeneous equation $Ly = 0$ we want to compute kernel of L , which is a set of ' k ' sequences: $a(n)$, $b(n)$, ... $z(n)$.

Solutions of L are linearly independent iff their Casoratian, denoted as $C(a, b, \dots, z)$, do not vanish for $n = 0$.

Casoratian is defined by $k \times k$ determinant:

$$\begin{vmatrix} + & a(n) & b(n) & \dots & z(n) & + \\ | & a(n+1) & b(n+1) & \dots & z(n+1) & | \\ | & \vdots & \vdots & \ddots & \vdots & | \\ | & \vdots & \vdots & \ddots & \vdots & | \\ + & a(n+k-1) & b(n+k-1) & \dots & z(n+k-1) & + \end{vmatrix}$$

It proves very useful in `rsolve_hyper()` where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```
>>> from sympy import Symbol, casoratian, factorial
>>> n = Symbol('n', integer=True)
```

Exponential and factorial are linearly independent:

```
>>> casoratian([2**n, factorial(n)], n) != 0
True
```

`sympy.matrices.dense.randMatrix`(r, c=None, min=0, max=99, seed=None, symmetric=False, percent=100, prng=None)

Create random matrix with dimensions $r \times c$. If c is omitted the matrix will be square. If symmetric is True the matrix must be square. If percent is less than 100 then only approximately the given percentage of elements will be non-zero.

The pseudo-random number generator used to generate matrix is chosen in the following way.

- If `prng` is supplied, it will be used as random number generator. It should be an instance of `random.Random`, or at least have `randint` and `shuffle` methods with same signatures.
- if `prng` is not supplied but `seed` is supplied, then new `random.Random` with given seed will be created;
- otherwise, a new `random.Random` with default seed will be used.

Examples

```
>>> from sympy.matrices import randMatrix
>>> randMatrix(3)
[25, 45, 27]
[44, 54, 9]
[23, 96, 46]
>>> randMatrix(3, 2)
[87, 29]
[23, 37]
[90, 26]
>>> randMatrix(3, 3, 0, 2)
[0, 2, 0]
[2, 0, 1]
[0, 0, 1]
>>> randMatrix(3, symmetric=True)
[85, 26, 29]
[26, 71, 43]
[29, 43, 57]
>>> A = randMatrix(3, seed=1)
>>> B = randMatrix(3, seed=2)
>>> A == B
False
>>> A == randMatrix(3, seed=1)
True
>>> randMatrix(3, symmetric=True, percent=50)
[77, 70, 0],
[70, 0, 0],
[0, 0, 88]
```

Numpy Utility Functions Reference

`sympy.matrices.dense.list2numpy(l, dtype=<class 'object'>)`
 Converts python list of SymPy expressions to a NumPy array.

See also:

[matrix2numpy](#) (page 759)

`sympy.matrices.dense.matrix2numpy(m, dtype=<class 'object'>)`
 Converts SymPy's matrix to a NumPy array.

See also:

[list2numpy](#) (page 759)

`sympy.matrices.dense.symarray(prefix, shape, **kwargs)`
 Create a numpy ndarray of symbols (as an object array).

The created symbols are named `prefix_i1_i2_...`. You should thus provide a non-empty prefix if you want your symbols to be unique for different output arrays, as SymPy symbols with identical names are the same object.

Parameters `prefix` : string

A prefix prepended to the name of every symbol.

shape : int or tuple

Shape of the created array. If an int, the array is one-dimensional; for more than one dimension the shape must be a tuple.

****kwargs** : dict
keyword arguments passed on to Symbol

Examples

These doctests require numpy.

```
>>> from sympy import symarray
>>> symarray('', 3)
[_0 _1 _2]
```

If you want multiple symarrays to contain distinct symbols, you must provide unique prefixes:

```
>>> a = symarray('', 3)
>>> b = symarray('', 3)
>>> a[0] == b[0]
True
>>> a = symarray('a', 3)
>>> b = symarray('b', 3)
>>> a[0] == b[0]
False
```

Creating symarrays with a prefix:

```
>>> symarray('a', 3)
[a_0 a_1 a_2]
```

For more than one dimension, the shape must be given as a tuple:

```
>>> symarray('a', (2, 3))
[[a_0_0 a_0_1 a_0_2]
 [a_1_0 a_1_1 a_1_2]]
>>> symarray('a', (2, 3, 2))
[[[a_0_0_0 a_0_0_1]
  [a_0_1_0 a_0_1_1]
  [a_0_2_0 a_0_2_1]]
 [a_1_0_0 a_1_0_1]
 [a_1_1_0 a_1_1_1]
 [a_1_2_0 a_1_2_1]]]
```

For setting assumptions of the underlying Symbols:

```
>>> [s.is_real for s in symarray('a', 2, real=True)]
[True, True]
```

`sympy.matrices.dense.rot_axis1(theta)`

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.

See also:

rot_axis2 (page 761) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

rot_axis3 (page 761) Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

Examples

```
>>> from sympy import pi
>>> from sympy.matrices import rot_axis1
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis1(theta)
Matrix([
[1,      0,      0],
[0,      1/2, sqrt(3)/2],
[0, -sqrt(3)/2,  1/2]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis1(pi/2)
Matrix([
[1,  0,  0],
[0,  0,  1],
[0, -1,  0]])
```

`sympy.matrices.dense.rot_axis2(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis.

See also:

rot_axis1 (page 760) Returns a rotation matrix for a rotation of θ (in radians) about the 1-axis

rot_axis3 (page 761) Returns a rotation matrix for a rotation of θ (in radians) about the 3-axis

Examples

```
>>> from sympy import pi
>>> from sympy.matrices import rot_axis2
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis2(theta)
Matrix([
[      1/2, 0, -sqrt(3)/2],
[      0, 1,      0],
[sqrt(3)/2, 0,      1/2]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis2(pi/2)
Matrix([
[0, 0, -1],
[0, 1,  0],
[1, 0,  0]])
```

`sympy.matrices.dense.rot_axis3(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 3-axis.

See also:

rot_axis1 (page 760) Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

rot_axis2 (page 761) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

Examples

```
>>> from sympy import pi
>>> from sympy.matrices import rot_axis3
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis3(theta)
Matrix([
[ 1/2, sqrt(3)/2, 0],
[-sqrt(3)/2, 1/2, 0],
[ 0, 0, 1]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis3(pi/2)
Matrix([
[ 0, 1, 0],
[-1, 0, 0],
[ 0, 0, 1]])
```

`sympy.matrices.matrices.a2idx(j, n=None)`
Return integer after making positive and validating against n.

5.17.2 Dense Matrices

Matrix Class Reference

`class sympy.matrices.dense.MutableDenseMatrix`

Attributes

cols	
rows	

`col_del(i)`
Delete the given column.

See also:

col, row_del (page 765)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.col_del(1)
>>> M
Matrix([
[1, 0],
[0, 0],
[0, 1]])
```

`col_op(j, f)`

In-place operation on col j using two-arg functor whose args are interpreted as $(self[i, j], i)$.

See also:

`col`, `row_op` (page 765)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([
[1, 2, 0],
[0, 1, 0],
[0, 0, 1]])
```

`col_swap(i, j)`

Swap the two given columns of the matrix in-place.

See also:

`col`, `row_swap` (page 765)

Examples

```
>>> from sympy.matrices import Matrix
>>> M = Matrix([[1, 0], [1, 0]])
>>> M
Matrix([
[1, 0],
[1, 0]])
>>> M.col_swap(0, 1)
>>> M
Matrix([
[0, 1],
[0, 1]])
```

`copyin_list(key, value)`

Copy in elements from a list.

Parameters `key` : slice

The section of this matrix to replace.

value : iterable

The iterable to copy values from.

See also:

[copyin_matrix](#) (page 764)

Examples

```
>>> from sympy.matrices import eye
>>> I = eye(3)
>>> I[:2, 0] = [1, 2] # col
>>> I
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
>>> I[1, :2] = [[3, 4]]
>>> I
Matrix([
[1, 0, 0],
[3, 4, 0],
[0, 0, 1]])
```

copyin_matrix(key, value)

Copy in values from a matrix into the given bounds.

Parameters **key** : slice

The section of this matrix to replace.

value : Matrix

The matrix to copy values from.

See also:

[copyin_list](#) (page 763)

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> M = Matrix([[0, 1], [2, 3], [4, 5]])
>>> I = eye(3)
>>> I[:3, :2] = M
>>> I
Matrix([
[0, 1, 0],
[2, 3, 0],
[4, 5, 1]])
>>> I[0, 1] = M
>>> I
Matrix([
[0, 0, 1],
[2, 2, 3],
[4, 4, 5]])
```

fill(value)

Fill the matrix with the scalar value.

See also:[zeros](#) (page 755), [ones](#) (page 755)**row_del**(i)

Delete the given row.

See also:row, [col_del](#) (page 762)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.row_del(1)
>>> M
Matrix([
[1, 0, 0],
[0, 0, 1]])
```

row_op(i, f)In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i, j], j).**See also:**row, [zip_row_op](#) (page 766), [col_op](#) (page 763)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
```

row_swap(i, j)

Swap the two given rows of the matrix in-place.

See also:row, [col_swap](#) (page 763)

Examples

```
>>> from sympy.matrices import Matrix
>>> M = Matrix([[0, 1], [1, 0]])
>>> M
Matrix([
[0, 1],
```

(continues on next page)

(continued from previous page)

```
[1, 0]])
>>> M.row_swap(0, 1)
>>> M
Matrix([
[1, 0],
[0, 1]])
```

simplify(ratio=1.7, measure=<function count_ops>)

Applies simplify to the elements of a matrix in place.

This is a shortcut for `M.applyfunc(lambda x: simplify(x, ratio, measure))`

See also:

[sympy.simplify.simplify.simplify](#) (page 1154)

zip_row_op(i, k, f)

In-place operation on row *i* using two-arg functor whose args are interpreted as `(self[i, j], self[k, j])`.

See also:

[row](#), [row_op](#) (page 765), [col_op](#) (page 763)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
```

ImmutableMatrix Class Reference

class `sympy.matrices.immutable.ImmutableDenseMatrix`

Create an immutable version of a matrix.

Examples

```
>>> from sympy import eye
>>> from sympy.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

is_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

Examples

```
>>> from sympy import Matrix, zeros
>>> from sympy.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero
```

5.17.3 Sparse Matrices

SparseMatrix Class Reference

class sympy.matrices.sparse.**SparseMatrix**

A sparse matrix (a matrix with a large number of zero elements).

See also:

sympy.matrices.dense.Matrix

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> SparseMatrix(2, 2, range(4))
Matrix([
[0, 1],
[2, 3]])
>>> SparseMatrix(2, 2, {(1, 1): 2})
Matrix([
[0, 0],
[0, 2]])
```

Attributes

cols	
rows	

CL

Alternate faster representation

LDLdecomposition()

Returns the LDL Decomposition (matrices L and D) of matrix A, such that $L * D * L.T == A$. A must be a square, symmetric, positive-definite and non-singular.

This method eliminates the use of square root and ensures that all the diagonal entries of L are 1.

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> A = SparseMatrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1,  0, 0],
[ 3/5, 1, 0],
[-1/5, 1/3, 1]])
>>> D
Matrix([
[25, 0, 0],
[ 0, 9, 0],
[ 0, 0, 9]])
>>> L * D * L.T == A
True
```

RL

Alternate faster representation

applyfunc(f)

Apply a function to each element of the matrix.

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> m = SparseMatrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([
[0, 1],
[2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([
[0, 2],
[4, 6]])
```

as_immutable()

Returns an Immutable version of this Matrix.

as_mutable()

Returns a mutable version of this matrix.

Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

cholesky()

Returns the Cholesky decomposition L of a matrix A such that $L * L.T = A$
 A must be a square, symmetric, positive-definite and non-singular matrix

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> A = SparseMatrix(((25,15,-5),(15,18,0),(-5,0,11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T == A
True
```

col_list()

Returns a column-sorted list of non-zero elements of the matrix.

See also:

`col_op`, `row_list` (page 770)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a=SparseMatrix((1, 2), (3, 4))
>>> a
Matrix([
[1, 2],
[3, 4]])
>>> a.CL
[(0, 0, 1), (1, 0, 3), (0, 1, 2), (1, 1, 4)]
```

liupc()

Liu's algorithm, for pre-determination of the Elimination Tree of the given matrix, used in row-based symbolic Cholesky factorization.

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Gron-delle (1999) <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582>

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix([
... [1, 0, 3, 2],
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.liupc()
([[0], [], [0], [1, 2]], [4, 3, 4, 4])
```

nnz()

Returns the number of non-zero elements in Matrix.

row_list()

Returns a row-sorted list of non-zero elements of the matrix.

See also:

row_op, col_list (page 769)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([
[1, 2],
[3, 4]])
>>> a.RL
[(0, 0, 1), (0, 1, 2), (1, 0, 3), (1, 1, 4)]
```

row_structure_symbolic_cholesky()

Symbolic cholesky factorization, for pre-determination of the non-zero structure of the Cholesky factorization.

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Gron-delle (1999) <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582>

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix([
... [1, 0, 3, 2],
```

(continues on next page)

(continued from previous page)

```
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.row_structure_symbolic_cholesky()
[[0], [], [0], [1, 2]]
```

scalar_multiply(scalar)

Scalar element-wise multiplication

solve(rhs, method='LDL')Return solution to $\text{self} * \text{soln} = \text{rhs}$ using given inversion method.For a list of possible inversion methods, see the `.inv()` docstring.**solve_least_squares(rhs, method='LDL')**

Return the least-square fit to the data.

By default the `cholesky_solve` routine is used (`method='CH'`); other methods of matrix inversion can be used. To find out which are available, see the docstring of the `.inv()` method.

Examples

```
>>> from sympy.matrices import SparseMatrix, Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = SparseMatrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of `S` represent coefficients of $Ax + By$ and x and y are $[2, 3]$ then $S*xy$ is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of `xy`:

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by $S*xy - r$:

```
>>> S*xy - r
Matrix([
[1/3],
[1/3],
[1/3]])
>>> _.norm().n(2)
0.58
```

If a different `xy` is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

class `sympy.matrices.sparse.MutableSparseMatrix`

Attributes

cols	
rows	

col_del(k)

Delete the given column of the matrix.

See also:

[row_del](#) (page 774)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix([[0, 0], [0, 1]])
>>> M
Matrix([
[0, 0],
[0, 1]])
>>> M.col_del(0)
>>> M
Matrix([
[0],
[1]])
```

col_join(other)

Returns B augmented beneath A (row-wise joining):

```
[A]
[B]
```

Examples

```
>>> from sympy import SparseMatrix, Matrix, ones
>>> A = SparseMatrix(ones(3))
>>> A
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
>>> B = SparseMatrix.eye(3)
>>> B
Matrix([
```

(continues on next page)

(continued from previous page)

```
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C = A.col_join(B); C
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1],
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C == A.col_join(Matrix(B))
True
```

Joining along columns is the same as appending rows at the end of the matrix:

```
>>> C == A.row_insert(A.rows, Matrix(B))
True
```

col_op(j, f)

In-place operation on col j using two-arg functor whose args are interpreted as $(\text{self}[i, j], i)$ for i in $\text{range}(\text{self.rows})$.

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[1, 0] = -1
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([
[ 2, 4, 0],
[-1, 0, 0],
[ 0, 0, 2]])
```

col_swap(i, j)

Swap, in place, columns i and j .

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.col_swap(1, 0); S
Matrix([
[0, 1, 0],
[1, 0, 0],
[2, 0, 1]])
```

fill(value)

Fill self with the given value.

Notes

Unless many values are going to be deleted (i.e. set to zero) this will create a matrix that is slower than a dense matrix in operations.

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.zeros(3); M
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
>>> M.fill(1); M
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
```

`row_del(k)`

Delete the given row of the matrix.

See also:

`col_del` (page 772)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix([[0, 0], [0, 1]])
>>> M
Matrix([
[0, 0],
[0, 1]])
>>> M.row_del(0)
>>> M
Matrix([[0, 1]])
```

`row_join(other)`

Returns B appended after A (column-wise augmenting):

```
[A B]
```

Examples

```
>>> from sympy import SparseMatrix, Matrix
>>> A = SparseMatrix(((1, 0, 1), (0, 1, 0), (1, 1, 0)))
>>> A
Matrix([
[1, 0, 1],
[0, 1, 0],
[1, 1, 0]])
```

(continues on next page)

(continued from previous page)

```

>>> B = SparseMatrix(((1, 0, 0), (0, 1, 0), (0, 0, 1)))
>>> B
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C = A.row_join(B); C
Matrix([
[1, 0, 1, 1, 0, 0],
[0, 1, 0, 0, 1, 0],
[1, 1, 0, 0, 0, 1]])
>>> C == A.row_join(Matrix(B))
True

```

Joining at row ends is the same as appending columns at the end of the matrix:

```

>>> C == A.col_insert(A.cols, B)
True

```

row_op(i, f)

In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i, j], j).

See also:

row, [zip_row_op](#) (page 775), [col_op](#) (page 773)

Examples

```

>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([
[2, -1, 0],
[4, 0, 0],
[0, 0, 2]])

```

row_swap(i, j)

Swap, in place, columns *i* and *j*.

Examples

```

>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.row_swap(1, 0); S
Matrix([
[0, 1, 0],
[1, 0, 0],
[0, 2, 1]])

```

zip_row_op(i, k, f)

In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i, j], self[k, j]).

See also:

`row`, `row_op` (page 775), `col_op` (page 773)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([
[2, -1, 0],
[4,  0, 0],
[0,  0, 2]])
```

ImmutableSparseMatrix Class Reference

class `sympy.matrices.immutable.ImmutableSparseMatrix`

Create an immutable version of a sparse matrix.

Examples

```
>>> from sympy import eye
>>> from sympy.matrices.immutable import ImmutableSparseMatrix
>>> ImmutableSparseMatrix(1, 1, {})
Matrix([[0]])
>>> ImmutableSparseMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableSparseMatrix
>>> _.shape
(3, 3)
```

Attributes

<code>cols</code>	
<code>rows</code>	

5.17.4 Immutable Matrices

The standard `Matrix` class in SymPy is mutable. This is important for performance reasons but means that standard matrices cannot interact well with the rest of SymPy. This is because the `Basic` object, from which most SymPy classes inherit, is immutable.

The mission of the `ImmutableDenseMatrix` class, which is aliased as `ImmutableMatrix` for short, is to bridge the tension between performance/mutability and safety/immutability. Immutable matrices can do almost everything that normal matrices can do but they inherit from `Basic` and can thus interact more naturally with the rest of SymPy. `ImmutableMatrix` also inherits from `MatrixExpr`, allowing it to interact freely with SymPy's Matrix Expression module.

You can turn any Matrix-like object into an `ImmutableMatrix` by calling the constructor

```
>>> from sympy import Matrix, ImmutableMatrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1, 1] = 0
>>> IM = ImmutableMatrix(M)
>>> IM
Matrix([
[1, 2, 3],
[4, 0, 6],
[7, 8, 9]])
>>> IM[1, 1] = 5
Traceback (most recent call last):
...
TypeError: Can not set values in Immutable Matrix. Use Matrix instead.
```

ImmutableMatrix Class Reference

class `sympy.matrices.immutable.ImmutableDenseMatrix`

Create an immutable version of a matrix.

Examples

```
>>> from sympy import eye
>>> from sympy.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

`is_zero`

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be `None`

Examples

```
>>> from sympy import Matrix, zeros
>>> from sympy.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero
```

5.17.5 Matrix Expressions

The Matrix expression module allows users to write down statements like

```
>>> from sympy import MatrixSymbol, Matrix
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).I*Y
 $X^{-1}X.T^{-1}Y$ 
```

```
>>> Matrix(X)
Matrix([
[X[0, 0], X[0, 1], X[0, 2]],
[X[1, 0], X[1, 1], X[1, 2]],
[X[2, 0], X[2, 1], X[2, 2]]])
```

```
>>> (X*Y)[1, 2]
 $X[1, 0]Y[0, 2] + X[1, 1]Y[1, 2] + X[1, 2]Y[2, 2]$ 
```

where X and Y are `MatrixSymbol` (page 780)'s rather than scalar symbols.

Matrix Expressions Core Reference

class sympy.matrices.expressions.**MatrixExpr**

Superclass for Matrix Expressions

MatrixExprs represent abstract matrices, linear transformations represented within a particular basis.

Examples

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).I * A * y
```


Attributes

is_Identity	

T

Matrix transposition.

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type ImmutableDenseMatrix.

See also:

as_mutable (page 779) returns mutable Matrix type

Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

as_mutable()

Returns a dense, mutable matrix with elements represented explicitly

See also:

as_explicit (page 779) returns ImmutableDenseMatrix

Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as_mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

equals(other)

Test elementwise equality between matrices, potentially of different types

```
>>> from sympy import Identity, eye
>>> Identity(3).equals(eye(3))
True
```

static `from_index_summation(expr, first_index=None, last_index=None)`

Parse expression of matrices with explicitly summed indices into a matrix expression without indices, if possible.

This transformation expressed in mathematical notation:

$$\sum_{j=0}^{N-1} A_{i,j} B_{j,k} \implies \mathbf{A} \cdot \mathbf{B}$$

Optional parameter `first_index`: specify which free index to use as the index starting the expression.

Examples

```
>>> from sympy import MatrixSymbol, MatrixExpr, Sum, Symbol
>>> from sympy.abc import i, j, k, l, N
>>> A = MatrixSymbol("A", N, N)
>>> B = MatrixSymbol("B", N, N)
>>> expr = Sum(A[i, j]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B
```

Transposition is detected:

```
>>> expr = Sum(A[j, i]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A.T*B
```

Detect the trace:

```
>>> expr = Sum(A[i, i], (i, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
Trace(A)
```

More complicated expressions:

```
>>> expr = Sum(A[i, j]*B[k, j]*A[l, k], (j, 0, N-1), (k, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B.T*A.T
```

class `sympy.matrices.expressions.MatrixSymbol`

Symbolic representation of a Matrix object

Creates a SymPy Symbol to represent a Matrix. This matrix has a shape and can be included in Matrix Expressions

```
>>> from sympy import MatrixSymbol, Identity
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

Attributes

is_Identity	

class sympy.matrices.expressions.**MatAdd**

A Sum of Matrix Expressions

MatAdd inherits from and operates like SymPy Add

```
>>> from sympy import MatAdd, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

Attributes

is_Identity	

class sympy.matrices.expressions.**MatMul**

A product of matrix expressions

Examples

```
>>> from sympy import MatMul, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

Attributes

is_Identity	

class sympy.matrices.expressions.**MatPow**

Attributes

is_Identity	

class sympy.matrices.expressions.**Inverse**

The multiplicative inverse of a matrix expression

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the inverse, use the `.inverse()` method of matrices.

Examples

```
>>> from sympy import MatrixSymbol, Inverse
>>> A = MatrixSymbol('A', 3, 3)
>>> B = MatrixSymbol('B', 3, 3)
>>> Inverse(A)
A-1
>>> A.inverse() == Inverse(A)
True
>>> (A*B).inverse()
B-1*A-1
>>> Inverse(A*B)
(A*B)-1
```

Attributes

is_Identity	

class sympy.matrices.expressions.Transpose

The transpose of a matrix expression.

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the transpose, use the `transpose()` function, or the `.T` attribute of matrices.

Examples

```
>>> from sympy.matrices import MatrixSymbol, Transpose
>>> from sympy.functions import transpose
>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)
>>> Transpose(A)
A.T
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B).T
>>> transpose(A*B)
B.T*A.T
```

Attributes

is_Identity	

class sympy.matrices.expressions.Trace

Matrix Trace

Represents the trace of a matrix expression.

```
>>> from sympy import MatrixSymbol, Trace, eye
>>> A = MatrixSymbol('A', 3, 3)
>>> Trace(A)
Trace(A)
```

See Also: `trace`

class sympy.matrices.expressions.**FunctionMatrix**

Represents a Matrix using a function (Lambda)

This class is an alternative to `SparseMatrix`

```
>>> from sympy import FunctionMatrix, symbols, Lambda, MatMul, Matrix
>>> i, j = symbols('i,j')
>>> X = FunctionMatrix(3, 3, Lambda((i, j), i + j))
>>> Matrix(X)
Matrix([
[0, 1, 2],
[1, 2, 3],
[2, 3, 4]])
```

```
>>> Y = FunctionMatrix(1000, 1000, Lambda((i, j), i + j))
```

```
>>> isinstance(Y*Y, MatMul) # this is an expression object
True
```

```
>>> (Y**2)[10,10] # So this is evaluated lazily
342923500
```

Attributes

is_Identity	

class sympy.matrices.expressions.**Identity**

The Matrix Identity I - multiplicative identity

```
>>> from sympy.matrices import Identity, MatrixSymbol
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

class sympy.matrices.expressions.**ZeroMatrix**

The Matrix Zero 0 - additive identity

```
>>> from sympy import MatrixSymbol, ZeroMatrix
>>> A = MatrixSymbol('A', 3, 5)
>>> Z = ZeroMatrix(3, 5)
>>> A+Z
A
>>> Z*A.T
0
```

Attributes

is_Identity	

Block Matrices

Block matrices allow you to construct larger matrices out of smaller sub-blocks. They can work with [MatrixExpr](#) (page 778) or `ImmutableMatrix` objects.

class `sympy.matrices.expressions.blockmatrix.BlockMatrix`

A `BlockMatrix` is a `Matrix` composed of other smaller, submatrices

The submatrices are stored in a SymPy `Matrix` object but accessed as part of a `Matrix Expression`

```
>>> from sympy import (MatrixSymbol, BlockMatrix, symbols,
...     Identity, ZeroMatrix, block_collapse)
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z*Y + Z]])
```

transpose()

Return transpose of matrix.

Examples

```
>>> from sympy import MatrixSymbol, BlockMatrix, ZeroMatrix
>>> from sympy.abc import l, m, n
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> B.transpose()
Matrix([
[X.T, 0],
[Z.T, Y.T]])
>>> _.transpose()
Matrix([
[X, Z],
[0, Y]])
```

class sympy.matrices.expressions.blockmatrix.**BlockDiagMatrix**

A BlockDiagMatrix is a BlockMatrix with matrices only along the diagonal

```
>>> from sympy import MatrixSymbol, BlockDiagMatrix, symbols, Identity
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> BlockDiagMatrix(X, Y)
Matrix([
[X, 0],
[0, Y]])
```

sympy.matrices.expressions.blockmatrix.**block_collapse**(expr)

Evaluates a block matrix expression

```
>>> from sympy import MatrixSymbol, BlockMatrix, symbols, Identity, Matrix, ZeroMatrix, block_collapse
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z*Y + Z]])
```

5.18 Polynomials Manipulation Module

Computations with polynomials are at the core of computer algebra and having a fast and robust polynomials manipulation module is a key for building a powerful symbolic manipulation system. SymPy has a dedicated module `sympy.polys` for computing in polynomial algebras over various coefficient domains.

There is a vast number of methods implemented, ranging from simple tools like polynomial division, to advanced concepts including Gröbner bases and multivariate factorization over algebraic number domains.

5.18.1 Contents

Basic functionality of the module

Introduction

This tutorial tries to give an overview of the functionality concerning polynomials within SymPy. All code examples assume:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
```

Basic concepts

Polynomials

Given a family (x_i) of symbols, or other suitable objects, including numbers, expressions derived from them by repeated addition, subtraction and multiplication are called polynomial expressions in the generators x_i .

By the distributive law it is possible to perform multiplications before additions and subtractions. The products of generators thus obtained are called monomials. They are usually written in the form $x_1^{\nu_1} x_2^{\nu_2} \cdots x_n^{\nu_n}$ where the exponents ν_i are nonnegative integers. It is often convenient to write this briefly as x^ν where $x = (x_1, x_2, \dots, x_n)$ denotes the family of generators and $\nu = (\nu_1, \nu_2, \dots, \nu_n)$ is the family of exponents.

When all monomials having the same exponents are combined, the polynomial expression becomes a sum of products $c_\nu x^\nu$, called the terms of the polynomial, where the coefficients c_ν are integers. If some of the x_i are manifest numbers, they are incorporated in the coefficients and not regarded as generators. Such coefficients are typically rational, real or complex numbers. Some symbolic numbers, e.g., `pi`, can be either coefficients or generators.

A polynomial expression that is a sum of terms with different monomials is uniquely determined by its family of coefficients (c_ν) . Such an expression is customarily called a polynomial, though, more properly, that name does stand for the coefficient family once the generators are given. SymPy implements polynomials by default as dictionaries with monomials as keys and coefficients as values. Another implementation consists of nested lists of coefficients.

The set of all polynomials with integer coefficients in the generators x_i is a ring, i.e., the sums, differences and products of its elements are again polynomials in the same generators. This ring is denoted $\mathbb{Z}[x_1, x_2, \dots, x_n]$, or $\mathbb{Z}[(x_i)]$, and called the ring of polynomials in the x_i with integer coefficients.

More generally, the coefficients of a polynomial can be elements of any commutative ring A , and the corresponding polynomial ring is then denoted $A[x_1, x_2, \dots, x_n]$. The ring A can also be a polynomial ring. In SymPy, the coefficient ring is called the `domain` of the polynomial ring, and it can be given as a keyword parameter. By default, it is determined by the coefficients of the polynomial arguments.

Polynomial expressions can be transformed into polynomials by the method `as_poly()`:

```
>>> e = (x + y)*(y - 2*z)
>>> e.as_poly()
Poly(x*y - 2*x*z + y**2 - 2*y*z, x, y, z, domain='ZZ')
```

If a polynomial expression contains numbers that are not integers, they are regarded as coefficients and the coefficient ring is extended accordingly. In particular, division by integers leads to rational coefficients:

```
>>> e = (3*x/2 + y)*(z - 1)
>>> e.as_poly()
Poly(3/2*x*z - 3/2*x + y*z - y, x, y, z, domain='QQ')
```


Symbolic numbers are considered generators unless they are explicitly excluded, in which case they are adjoined to the coefficient ring:

```
>>> e = (x + 2*pi)*y
>>> e.as_poly()
Poly(x*y + 2*y*pi, x, y, pi, domain='ZZ')
>>> e.as_poly(x, y)
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Alternatively, the coefficient domain can be specified by means of a keyword argument:

```
>>> e = (x + 2*pi)*y
>>> e.as_poly(domain=ZZ[pi])
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Note that the ring $\mathbb{Z}[\pi][x, y]$ of polynomials in x and y with coefficients in $\mathbb{Z}[\pi]$ is mathematically equivalent to $\mathbb{Z}[\pi, x, y]$, only their implementations differ.

If an expression contains functions of the generators, other than their positive integer powers, these are interpreted as new generators:

```
>>> e = x*sin(y) - y
>>> e.as_poly()
Poly(x*(sin(y)) - y, x, y, sin(y), domain='ZZ')
```

Since y and $\sin(y)$ are algebraically independent they can both appear as generators in a polynomial. However, polynomial expressions must not contain negative powers of generators:

```
>>> e = x - 1/x
>>> e.as_poly()
Poly(x - (1/x), x, 1/x, domain='ZZ')
```

It is important to realize that the generators x and $1/x = x^{-1}$ are treated as algebraically independent variables. In particular, their product is not equal to 1. Hence generators in denominators should be avoided even if they raise no error in the current implementation. This behavior is undesirable and may change in the future. Similar problems emerge with rational powers of generators. So, for example, x and $\sqrt{x} = x^{1/2}$ are not recognized as algebraically dependent.

If there are algebraic numbers in an expression, it is possible to adjoin them to the coefficient ring by setting the keyword `extension`:

```
>>> e = x + sqrt(2)
>>> e.as_poly()
Poly(x + (sqrt(2)), x, sqrt(2), domain='ZZ')
>>> e.as_poly(extension=True)
Poly(x + sqrt(2), x, domain='QQ<sqrt(2)>')
```

With the default setting `extension=False`, both x and $\sqrt{2}$ are incorrectly considered algebraically independent variables. With coefficients in the extension field $\mathbb{Q}(\sqrt{2})$ the square root is treated properly as an algebraic number. Setting `extension=True` whenever algebraic numbers are involved is definitely recommended even though it is not forced in the current implementation.

Divisibility

The fourth rational operation, division, or inverted multiplication, is not generally possible in rings. If a and b are two elements of a ring A , then there may exist a third element q in A such that $a = bq$. In fact, there may exist several such elements.

If also $a = bq'$ for some q' in A , then $b(q - q') = 0$. Hence either b or $q - q'$ is zero, or they are both zero divisors, nonzero elements whose product is zero.

Integral domains

Commutative rings with no zero divisors are called integral domains. Most of the commonly encountered rings, the ring of integers, fields, and polynomial rings over integral domains are integral domains.

Assume now that A is an integral domain, and consider the set P of its nonzero elements, which is closed under multiplication. If a and b are in P , and there exists an element q in P such that $a = bq$, then q is unique and called the quotient, a/b , of a by b . Moreover, it is said that

- a is divisible by b ,
- b is a divisor of a ,
- a is a multiple of b ,
- b is a factor of a .

An element a of P is a divisor of 1 if and only if it is invertible in A , with the inverse $a^{-1} = 1/a$. Such elements are called units. The units of the ring of integers are 1 and -1 . The invertible elements in a polynomial ring over a field are the nonzero constant polynomials.

If two elements of P , a and b , are divisible by each other, then the quotient a/b is invertible with inverse b/a , or equivalently, $b = ua$ where u is a unit. Such elements are said to be associated with, or associates of, each other. The associates of an integer n are n and $-n$. In a polynomial ring over a field the associates of a polynomial are its constant multiples.

Each element of P is divisible by its associates and the units. An element is irreducible if it has no other divisors and is not a unit. The irreducible elements in the ring of integers are the prime numbers p and their opposites $-p$. In a field, every nonzero element is invertible and there are no irreducible elements.

Factorial domains

In the ring of integers, each nonzero element can be represented as a product of irreducible elements and optionally a unit ± 1 . Moreover, any two such products have the same number of irreducible factors which are associated with each other in a suitable order. Integral domains having this property are called factorial, or unique factorization domains. In addition to the ring of integers, all polynomial rings over a field are factorial, and so are more generally polynomial rings over any factorial domain. Fields are trivially factorial since there are only units. The irreducible elements of a factorial domain are usually called primes.

A family of integers has only a finite number of common divisors and the greatest of them is divisible by all of them. More generally, given a family of nonzero elements (a_i) in an integral domain, a common divisor d of the elements is called a greatest common divisor, abbreviated gcd, of the family if it is a multiple of all common divisors. A greatest common divisor, if it exists, is not unique in general; all of its associates have the same property. It is denoted by

$d = \gcd(a_1, \dots, a_n)$ if there is no danger of confusion. A least common multiple, or lcm, of a family (a_i) is defined analogously as a common multiple m that divides all common multiples. It is denoted by $m = \text{lcm}(a_1, \dots, a_n)$.

In a factorial domain, greatest common divisors always exists. They can be found, at least in principle, by factoring each element of a family into a product of prime powers and an optional unit, and, for each prime, taking the least power that appears in the factorizations. The product of these prime powers is then a greatest common divisor. A least common multiple can be obtained from the same factorizations as the product of the greatest powers for each prime.

Euclidean domains

A practical algorithm for computing a greatest common divisor can be implemented in Euclidean domains. They are integral domains that can be endowed with a function w assigning a nonnegative integer to each nonzero element of the domain and having the following property:

if a and b are nonzero, there are q and r that satisfy the division identity

$$a = qb + r$$

such that either $r = 0$ or $w(r) < w(b)$.

The ring of integers and all univariate polynomial rings over fields are Euclidean domains with $w(a) = |a|$ resp. $w(a) = \deg(a)$.

The division identity for integers is implemented in Python as the built-in function `divmod()` that can also be applied to SymPy Integers:

```
>>> divmod(Integer(53), Integer(7))
(7, 4)
```

For polynomials the division identity is given in SymPy by the function `div()`:

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x  5
--- + -
 2    2
>>> r
-2
>>> (q*g + r).expand()
      2
5*x  + 10*x + 3
```

The division identity can be used to determine the divisibility of elements in a Euclidean domain. If $r = 0$ in the division identity, then a is divisible by b . Conversely, if $a = cb$ for some element c , then $(c - q)b = r$. It follows that $c = q$ and $r = 0$ if w has the additional property:

if a and b are nonzero, then $w(ab) \geq w(b)$.

This is satisfied by the functions given above. (And it is always possible to redefine $w(a)$ by taking the minimum of the values $w(xa)$ for $x \neq 0$.)

The principal application of the division identity is the efficient computation of a greatest common divisor by means of the [Euclidean algorithm](#). It applies to two elements of a Euclidean

domain. A gcd of several elements can be obtained by iteration.

The function for computing the greatest common divisor of integers in SymPy is currently `igcd()`:

```
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

For univariate polynomials over a field the function has its common name `gcd()`, and the returned polynomial is monic:

```
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g, domain=QQ)
x + 1/2
```

Divisibility of polynomials

The ring $A = \mathbb{Z}[x]$ of univariate polynomials over the ring of integers is not Euclidean but it is still factorial. To see this, consider the divisibility in A .

Let f and g be two nonzero polynomials in A . If f is divisible by g in A , then it is also divisible in the ring $B = \mathbb{Q}[x]$ of polynomials with rational coefficients. Since B is Euclidean, this can be determined by means of the division identity.

Assume, conversely, that $f = gh$ for some polynomial h in B . Then f is divisible by g in A if and only if the coefficients of h are integers. To find out when this is true it is necessary to consider the divisibility of the coefficients.

For a polynomial f in A , let c be the greatest common divisor of its coefficients. Then f is divisible by the constant polynomial c in A , and the quotient $f/c = p$ is a polynomial whose coefficients are integers that have no common divisor apart from the units. Such polynomials are called primitive. A polynomial with rational coefficients can also be written as $f = cp$, where c is a rational number and p is a primitive polynomial. The constant c is called the content of f , and p is its primitive part. These components can be found by the method `as_content_primitive()`:

```
>>> f = 6*x**2 - 3*x + 9
>>> c, p = f.as_content_primitive()
>>> c, p
2
(3, 2*x**2 - x + 3)
>>> f = x**2/3 - x/2 + 1
>>> c, p = f.as_content_primitive()
>>> c, p
2
(1/6, 2*x**2 - 3*x + 6)
```

Let f, f' be polynomials with contents c, c' and primitive parts p, p' . Then $ff' = (cc')(pp')$ where the product pp' is primitive by [Gauss's lemma](#). It follows that

the content of a product of polynomials is the product of their contents and the primitive part of the product is the product of the primitive parts.

Returning to the divisibility in the ring $\mathbb{Z}[x]$, assume that f and g are two polynomials with integer coefficients such that the division identity in $\mathbb{Q}[x]$ yields the equality $f = gh$ for some

polynomial h with rational coefficients. Then the content of f is equal to the content of g multiplied by the content of h . As h has integer coefficients if and only if its content is an integer, we get the following criterion:

f is divisible by g in the ring $\mathbb{Z}[x]$ if and only if

1. f is divisible by g in $\mathbb{Q}[x]$, and
2. the content of f is divisible by the content of g in $\mathbb{Z}$.

If $f = cp$ is irreducible in $\mathbb{Z}[x]$, then either c or p must be a unit. If p is not a unit, it must be irreducible also in $\mathbb{Q}[x]$. For if it is a product of two polynomials, it is also the product of their primitive parts, and one of them must be a unit. Hence there are two kinds of irreducible elements in $\mathbb{Z}[x]$:

1. prime numbers of $\mathbb{Z}$, and
2. primitive polynomials that are irreducible in $\mathbb{Q}[x]$.

It follows that each polynomial in $\mathbb{Z}[x]$ is a product of irreducible elements. It suffices to factor its content and primitive part separately. These products are essentially unique; hence $\mathbb{Z}[x]$ is also factorial.

Another important consequence is that a greatest common divisor of two polynomials in $\mathbb{Z}[x]$ can be found efficiently by applying the Euclidean algorithm separately to their contents and primitive parts in the Euclidean domains $\mathbb{Z}$ and $\mathbb{Q}[x]$. This is also implemented in SymPy:

```
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g)
2*x + 1
>>> gcd(6*f, 3*g)
6*x + 3
```

Basic functionality

These functions provide different algorithms dealing with polynomials in the form of SymPy expression, like symbols, sums etc.

Division

The function `div()` provides division of polynomials with remainder. That is, for polynomials f and g , it computes q and r , such that $f = g \cdot q + r$ and $\deg(r) < \deg(g)$. For polynomials in one variable with coefficients in a field, say, the rational numbers, q and r are uniquely defined this way:

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x  5
--- + -
2    2
>>> r
-2
>>> (q*g + r).expand()
```

(continues on next page)

(continued from previous page)

$$5x^2 + 10x + 3$$

As you can see, q has a non-integer coefficient. If you want to do division only in the ring of polynomials with integer coefficients, you can specify an additional parameter:

```
>>> q, r = div(f, g, domain='ZZ')
>>> q
0
>>> r
      2
5*x  + 10*x + 3
```

But be warned, that this ring is no longer Euclidean and that the degree of the remainder doesn't need to be smaller than that of f . Since 2 doesn't divide 5, $2x$ doesn't divide $5x^2$, even if the degree is smaller. But:

```
>>> g = 5*x + 1
>>> q, r = div(f, g, domain='ZZ')
>>> q
x
>>> r
9*x + 3
>>> (q*g + r).expand()
      2
5*x  + 10*x + 3
```

This also works for polynomials with multiple variables:

```
>>> f = x*y + y*z
>>> g = 3*x + 3*z
>>> q, r = div(f, g, domain='QQ')
>>> q
y
-
3
>>> r
0
```

In the last examples, all of the three variables x , y and z are assumed to be variables of the polynomials. But if you have some unrelated constant as coefficient, you can specify the variables explicitly:

```
>>> a, b, c = symbols('a,b,c')
>>> f = a*x**2 + b*x + c
>>> g = 3*x + 2
>>> q, r = div(f, g, domain='QQ')
>>> q
a*x  2*a  b
--- - --- + -
 3      9   3
>>> r
4*a  2*b
```

(continues on next page)

(continued from previous page)

```

--- - --- + c
9      3

```

GCD and LCM

With division, there is also the computation of the greatest common divisor and the least common multiple.

When the polynomials have integer coefficients, the contents' gcd is also considered:

```

>>> f = (12*x + 12)*x
>>> g = 16*x**2
>>> gcd(f, g)
4*x

```

But if the polynomials have rational coefficients, then the returned polynomial is monic:

```

>>> f = 3*x**2/2
>>> g = 9*x/4
>>> gcd(f, g)
x

```

It also works with multiple variables. In this case, the variables are ordered alphabetically, by default, which has influence on the leading coefficient:

```

>>> f = x*y/2 + y**2
>>> g = 3*x + 6*y

>>> gcd(f, g)
x + 2*y

```

The lcm is connected with the gcd and one can be computed using the other:

```

>>> f = x*y**2 + x**2*y
>>> g = x**2*y**2
>>> gcd(f, g)
x*y
>>> lcm(f, g)
3 2 2 3
x *y + x *y
>>> (f*g).expand()
4 3 3 4
x *y + x *y
>>> (gcd(f, g, x, y)*lcm(f, g, x, y)).expand()
4 3 3 4
x *y + x *y

```

Square-free factorization

The square-free factorization of a univariate polynomial is the product of all factors (not necessarily irreducible) of degree 1, 2 etc.:

```
>>> f = 2*x**2 + 5*x**3 + 4*x**4 + x**5
>>> sqf_list(f)
(1, [(x + 2, 1), (x, 2), (x + 1, 2)])
>>> sqf(f)
      2      2
x *(x + 1) *(x + 2)
```

Factorization

This function provides factorization of univariate and multivariate polynomials with rational coefficients:

```
>>> factor(x**4/2 + 5*x**3/12 - x**2/3)
      2
x *(2*x - 1)*(3*x + 4)
-----
      12
>>> factor(x**2 + 4*x*y + 4*y**2)
      2
(x + 2*y)
```

Groebner bases

Buchberger's algorithm is implemented, supporting various monomial orders:

```
>>> groebner([x**2 + 1, y**4*x + x**3], x, y, order='lex')
      / [ 2      4      ] \
GroebnerBasis\[x + 1, y - 1], x, y, domain=ZZ, order=lex/

>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], x, y, z, order='grevlex')
      / [ 4      3      2      ] \
GroebnerBasis\[y - 1, z , x + 1], x, y, z, domain=ZZ, order=grevlex/
```

Solving Equations

We have (incomplete) methods to find the complex or even symbolic roots of polynomials and to solve some systems of polynomial equations:

```
>>> from sympy import roots, solve_poly_system
>>> solve(x**3 + 2*x + 3, x)
      1      \ / 11 *I      1      \ / 11 *I
[-1, - - - - - , - + - - - -]
      2      2      2      2

>>> p = Symbol('p')
```

(continues on next page)

(continued from previous page)

```

>>> q = Symbol('q')
>>> solve(x**2 + p*x + q, x)

```

$$\left[-\frac{p}{2} - \frac{\sqrt{p^2 - 4q}}{2}, -\frac{p}{2} + \frac{\sqrt{p^2 - 4q}}{2} \right]$$

```

>>> solve_poly_system([y - x, x - 5], x, y)
[(5, 5)]
>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)

```

$$[(0, -I), (0, I), (1, 0), \left(-\frac{1}{2} - \frac{\sqrt{3}I}{2}, 0\right), \left(-\frac{1}{2} + \frac{\sqrt{3}I}{2}, 0\right)]$$

Examples from Wester's Article

Introduction

In this tutorial we present examples from Wester's article concerning comparison and critique of mathematical abilities of several computer algebra systems (see [Wester1999] (page 1866)). All the examples are related to polynomial and algebraic computations and SymPy specific remarks were added to all of them.

Examples

All examples in this tutorial are computable, so one can just copy and paste them into a Python shell and do something useful with them. All computations were done using the following setup:

```

>>> from sympy import *
>>> init_printing(use_unicode=True, wrap_line=False, no_global=True)
>>> var('x,y,z,s,c,n')
(x, y, z, s, c, n)

```

Simple univariate polynomial factorization

To obtain a factorization of a polynomial use `factor()` function. By default `factor()` returns the result in unevaluated form, so the content of the input polynomial is left unexpanded, as in the following example:

```

>>> factor(6*x - 10)
2*(3*x - 5)

```

To achieve the same effect in a more systematic way use `primitive()` function, which returns the content and the primitive part of the input polynomial:

```
>>> primitive(6*x - 10)
(2, 3*x - 5)
```

Note: The content and the primitive part can be computed only over a ring. To simplify coefficients of a polynomial over a field use `monic()`.

Univariate GCD, resultant and factorization

Consider univariate polynomials f , g and h over integers:

```
>>> f = 64*x**34 - 21*x**47 - 126*x**8 - 46*x**5 - 16*x**60 - 81
>>> g = 72*x**60 - 25*x**25 - 19*x**23 - 22*x**39 - 83*x**52 + 54*x**10 + 81
>>> h = 34*x**19 - 25*x**16 + 70*x**7 + 20*x**3 - 91*x - 86
```

We can compute the greatest common divisor (GCD) of two polynomials using `gcd()` function:

```
>>> gcd(f, g)
1
```

We see that f and g have no common factors. However, $f*h$ and $g*h$ have an obvious factor h :

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

The same can be verified using the resultant of univariate polynomials:

```
>>> resultant(expand(f*h), expand(g*h))
0
```

Factorization of large univariate polynomials (of degree 120 in this case) over integers is also possible:

```
>>> factor(expand(f*g))
( 60      47      34      8      5      ) ( 60      52      39      25
 23      10      )
- ( 16*x  + 21*x  - 64*x  + 126*x  + 46*x  + 81 ) * ( 72*x  - 83*x  - 22*x  - 25*x  -
 19*x  + 54*x  + 81 )
```

Multivariate GCD and factorization

What can be done in univariate case, can be also done for multivariate polynomials. Consider the following polynomials f , g and h in $\mathbb{Z}[x, y, z]$:

```
>>> f = 24*x*y**19*z**8 - 47*x**17*y**5*z**8 + 6*x**15*y**9*z**2 - 3*x**22 + 5
>>> g = 34*x**5*y**8*z**13 + 20*x**7*y**7*z**7 + 12*x**9*y**16*z**4 + 80*y**14*z
>>> h = 11*x**12*y**7*z**13 - 23*x**2*y**8*z**10 + 47*x**17*y**5*z**8
```

As previously, we can verify that f and g have no common factors:

```
>>> gcd(f, g)
1
```

However, $f \cdot h$ and $g \cdot h$ have an obvious factor h :

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

Multivariate factorization of large polynomials is also possible:

```
>>> factor(expand(f*g))
      7      9      9      3      7      6      5      12      7      22      17      5      8      15
      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2      ↪ 2
-2·y ·z·(6·x ·y ·z + 10·x ·z + 17·x ·y·z + 40·y )·(3·x + 47·x ·y ·z - 6·x ·y·z
↪ ·z - 24·x·y ·z - 5)
```

Support for symbols in exponents

Polynomial manipulation functions provided by `sympy.polys` are mostly used with integer exponents. However, it's perfectly valid to compute with symbolic exponents, e.g.:

```
>>> gcd(2*x**(n + 4) - x**(n + 2), 4*x**(n + 1) + 3*x**n)
n
x
```

Testing if polynomials have common zeros

To test if two polynomials have a root in common we can use `resultant()` function. The theory says that the resultant of two polynomials vanishes if there is a common zero of those polynomials. For example:

```
>>> resultant(3*x**4 + 3*x**3 + x**2 - x - 2, x**3 - 3*x**2 + x + 5)
0
```

We can visualize this fact by factoring the polynomials:

```
>>> factor(3*x**4 + 3*x**3 + x**2 - x - 2)
      3
(x + 1)·(3·x + x - 2)

>>> factor(x**3 - 3*x**2 + x + 5)
      2
(x + 1)·(x - 4·x + 5)
```

In both cases we obtained the factor $x + 1$ which tells us that the common root is $x = -1$.

Normalizing simple rational functions

To remove common factors from the numerator and the denominator of a rational function the elegant way, use `cancel()` function. For example:

```
>>> cancel((x**2 - 4)/(x**2 + 4*x + 4))
x - 2
-----
x + 2
```

Expanding expressions and factoring back

One can work easily with expressions in both expanded and factored forms. Consider a polynomial f in expanded form. We differentiate it and factor the result back:

```
>>> f = expand((x + 1)**20)
>>> g = diff(f, x)
>>> factor(g)
19
20·(x + 1)
```

The same can be achieved in factored form:

```
>>> diff((x + 1)**20, x)
19
20·(x + 1)
```

Factoring in terms of cyclotomic polynomials

SymPy can very efficiently decompose polynomials of the form $x^n \pm 1$ in terms of cyclotomic polynomials:

```
>>> factor(x**15 - 1)
(x - 1)·(x2 + x + 1)·(x4 + x3 + x2 + x + 1)·(x8 - x7 + x5 - x4 + x3 - x2 + x + 1)
```

The original Wester's example was $x^{100} - 1$, but was truncated for readability purpose. Note that this is not a big struggle for `factor()` to decompose polynomials of degree 1000 or greater.

Univariate factoring over Gaussian numbers

Consider a univariate polynomial f with integer coefficients:

```
>>> f = 4*x**4 + 8*x**3 + 77*x**2 + 18*x + 153
```

We want to obtain a factorization of f over Gaussian numbers. To do this we use `factor()` as previously, but this time we set `gaussian` keyword to `True`:

```
>>> factor(f, gaussian=True)
4·(x - 3·i/2)·(x + 3·i/2)·(x + 1 - 4·i)·(x + 1 + 4·i)
```

As the result we got a splitting factorization of f with monic factors (this is a general rule when computing in a field with SymPy). The `gaussian` keyword is useful for improving code readability, however the same result can be computed using more general syntax:

```
>>> factor(f, extension=I)
4·(x - 3·i/2)·(x + 3·i/2)·(x + 1 - 4·i)·(x + 1 + 4·i)
```

Computing with automatic field extensions

Consider two univariate polynomials f and g :

```
>>> f = x**3 + (sqrt(2) - 2)*x**2 - (2*sqrt(2) + 3)*x - 3*sqrt(2)
>>> g = x**2 - 2
```

We would like to reduce degrees of the numerator and the denominator of a rational function f/g . To do this we employ `cancel()` function:

```
>>> cancel(f/g)
      3      2      2
x  - 2·x  + √2·x  - 3·x - 2·√2·x - 3·√2
-----
              2
              x  - 2
```

Unfortunately nothing interesting happened. This is because by default SymPy treats $\sqrt{2}$ as a generator, obtaining a bivariate polynomial for the numerator. To make `cancel()` recognize algebraic properties of $\sqrt{2}$, one needs to use `extension` keyword:

```
>>> cancel(f/g, extension=True)
      2
x  - 2·x - 3
-----
x  - √2
```

Setting `extension=True` tells `cancel()` to find minimal algebraic number domain for the coefficients of f/g . The automatically inferred domain is $\mathbb{Q}(\sqrt{2})$. If one doesn't want to rely on automatic inference, the same result can be obtained by setting the `extension` keyword with an explicit algebraic number:

```
>>> cancel(f/g, extension=sqrt(2))
      2
x  - 2·x - 3
-----
x  - √2
```

Univariate factoring over various domains

Consider a univariate polynomial f with integer coefficients:

```
>>> f = x**4 - 3*x**2 + 1
```

With `sympy.polys` we can obtain factorizations of f over different domains, which includes:

- rationals:

```
>>> factor(f)
      2      2
( x  - x - 1 ) · ( x  + x - 1 )
```

- finite fields:

```
>>> factor(f, modulus=5)
      2      2
(x - 2) · (x + 2)
```

- algebraic numbers:

```
>>> alg = AlgebraicNumber((sqrt(5) - 1)/2, alias='alpha')

>>> factor(f, extension=alg)
(x - α) · (x + α) · (x - 1 - α) · (x + α + 1)
```

Factoring polynomials into linear factors

Currently SymPy can factor polynomials into irreducibles over various domains, which can result in a splitting factorization (into linear factors). However, there is currently no systematic way to infer a splitting field (algebraic number field) automatically. In future the following syntax will be implemented:

```
>>> factor(x**3 + x**2 - 7, split=True)
Traceback (most recent call last):
...
NotImplementedError: 'split' option is not implemented yet
```

Note this is different from `extension=True`, because the later only tells how expression parsing should be done, not what should be the domain of computation. One can simulate the `split` keyword for several classes of polynomials using `solve()` function.

Advanced factoring over finite fields

Consider a univariate polynomial f with integer coefficients:

```
>>> f = x**11 + x + 1
```

We can factor f over a large finite field F_{65537} :

```
>>> factor(f, modulus=65537)
      2      9      8      6      5      3      2
(x + x + 1) · (x - x + x - x + x - x + 1)
```

and expand the resulting factorization back:

```
>>> expand(_)
      11
x  + x + 1
```

obtaining polynomial f . This was done using symmetric polynomial representation over finite fields. The same thing can be done using non-symmetric representation:

```
>>> factor(f, modulus=65537, symmetric=False)
      2      9      8      6      5      3      2
(x + x + 1) · (x + 65536·x + x + 65536·x + x + 65536·x + 1)
```

As with symmetric representation we can expand the factorization to get the input polynomial back. This time, however, we need to truncate coefficients of the expanded polynomial modulo 65537:

```
>>> trunc(expand(_), 65537)
11
x  + x + 1
```

Working with expressions as polynomials

Consider a multivariate polynomial f in $\mathbb{Z}[x, y, z]$:

```
>>> f = expand((x - 2*y**2 + 3*z**3)**20)
```

We want to compute factorization of f . To do this we use `factor` as usually, however we note that the polynomial in consideration is already in expanded form, so we can tell the factorization routine to skip expanding f :

```
>>> factor(f, expand=False)
      20
(
  2      3
(x - 2·y  + 3·z )
)
```

The default in `sympy.polys` is to expand all expressions given as arguments to polynomial manipulation functions and `Poly` class. If we know that expanding is unnecessary, then by setting `expand=False` we can save quite a lot of time for complicated inputs. This can be really important when computing with expressions like:

```
>>> g = expand((sin(x) - 2*cos(y)**2 + 3*tan(z)**3)**20)
>>> factor(g, expand=False)
      20
(
  2      3
(-sin(x) + 2·cos (y) - 3·tan (z))
)
```

Computing reduced Gröbner bases

To compute a reduced Gröbner basis for a set of polynomials use `groebner()` function. The function accepts various monomial orderings, e.g.: `lex`, `grlex` and `grevlex`, or a user defined one, via `order` keyword. The `lex` ordering is the most interesting because it has elimination property, which means that if the system of polynomial equations to `groebner()` is zero-dimensional (has finite number of solutions) the last element of the basis is a univariate polynomial. Consider the following example:

```
>>> f = expand((1 - c**2)**5 * (1 - s**2)**5 * (c**2 + s**2)**10)
>>> groebner([f, c**2 + s**2 - 1])
      2      2      20      18      16      14      12      10
GroebnerBasis([c  + s  - 1, c  - 5·c  + 10·c  - 10·c  + 5·c  - c ], s, c,
               domain=ℤ, order=lex)
```

The result is an ordinary Python list, so we can easily apply a function to all its elements, for example we can factor those elements:

```
>>> list(map(factor, _))
[ 2      2      10      5      5]
[c + s - 1, c - (c - 1) - (c + 1)]
```

From the above we can easily find all solutions of the system of polynomial equations. Or we can use `solve()` to achieve this in a more systematic way:

```
>>> solve([f, s**2 + c**2 - 1], c, s)
[(-1, 0), (0, -1), (0, 1), (1, 0)]
```

Multivariate factoring over algebraic numbers

Computing with multivariate polynomials over various domains is as simple as in univariate case. For example consider the following factorization over $\mathbb{Q}(\sqrt{-3})$:

```
>>> factor(x**3 + y**3, extension=sqrt(-3))
(x + y) * (x + y * (1/2 - sqrt(3)*I/2)) * (x + y * (1/2 + sqrt(3)*I/2))
```

Note: Currently multivariate polynomials over finite fields aren't supported.

Partial fraction decomposition

Consider a univariate rational function `f` with integer coefficients:

```
>>> f = (x**2 + 2*x + 3)/(x**3 + 4*x**2 + 5*x + 2)
```

To decompose `f` into partial fractions use `apart()` function:

```
>>> apart(f)
3      2      2
--- - --- + ---
x + 2  x + 1  (x + 1)
```

To return from partial fractions to the rational function use a composition of `together()` and `cancel()`:

```
>>> cancel(together(_))
2
x + 2*x + 3
-----
3      2
x + 4*x + 5*x + 2
```


Literature

Polynomials Manipulation Module Reference

Basic polynomial manipulation functions

`sympy.polys.polytools.poly(expr, *gens, **args)`
Efficiently transform an expression into a polynomial.

Examples

```
>>> from sympy import poly
>>> from sympy.abc import x
```

```
>>> poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

`sympy.polys.polytools.poly_from_expr(expr, *gens, **args)`
Construct a polynomial from an expression.

`sympy.polys.polytools.parallel_poly_from_expr(exprs, *gens, **args)`
Construct polynomials from expressions.

`sympy.polys.polytools.degree(f, gen=0)`
Return the degree of f in the given variable.
The degree of 0 is negative infinity.

See also:

`total_degree`, `degree_list` (page 803)

Examples

```
>>> from sympy import degree
>>> from sympy.abc import x, y
```

```
>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-oo
```

`sympy.polys.polytools.degree_list(f, *gens, **args)`
Return a list of degrees of f in all variables.

Examples

```
>>> from sympy import degree_list
>>> from sympy.abc import x, y
```

```
>>> degree_list(x**2 + y*x + 1)
(2, 1)
```

`sympy.polys.polytools.LC(f, *gens, **args)`
Return the leading coefficient of `f`.

Examples

```
>>> from sympy import LC
>>> from sympy.abc import x, y
```

```
>>> LC(4*x**2 + 2*x*y**2 + x*y + 3*y)
4
```

`sympy.polys.polytools.LM(f, *gens, **args)`
Return the leading monomial of `f`.

Examples

```
>>> from sympy import LM
>>> from sympy.abc import x, y
```

```
>>> LM(4*x**2 + 2*x*y**2 + x*y + 3*y)
x**2
```

`sympy.polys.polytools.LT(f, *gens, **args)`
Return the leading term of `f`.

Examples

```
>>> from sympy import LT
>>> from sympy.abc import x, y
```

```
>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

`sympy.polys.polytools.pdiv(f, g, *gens, **args)`
Compute polynomial pseudo-division of `f` and `g`.

Examples

```
>>> from sympy import pdiv
>>> from sympy.abc import x
```

```
>>> pdiv(x**2 + 1, 2*x - 4)
(2*x + 4, 20)
```

`sympy.polys.polytools.prem(f, g, *gens, **args)`
Compute polynomial pseudo-remainder of `f` and `g`.

Examples

```
>>> from sympy import prem
>>> from sympy.abc import x
```

```
>>> prem(x**2 + 1, 2*x - 4)
20
```

`sympy.polys.polytools.pquo(f, g, *gens, **args)`
 Compute polynomial pseudo-quotient of f and g .

Examples

```
>>> from sympy import pquo
>>> from sympy.abc import x
```

```
>>> pquo(x**2 + 1, 2*x - 4)
2*x + 4
>>> pquo(x**2 - 1, 2*x - 1)
2*x + 1
```

`sympy.polys.polytools.pexquo(f, g, *gens, **args)`
 Compute polynomial exact pseudo-quotient of f and g .

Examples

```
>>> from sympy import pexquo
>>> from sympy.abc import x
```

```
>>> pexquo(x**2 - 1, 2*x - 2)
2*x + 2
```

```
>>> pexquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.div(f, g, *gens, **args)`
 Compute polynomial division of f and g .

Examples

```
>>> from sympy import div, ZZ, QQ
>>> from sympy.abc import x
```

```
>>> div(x**2 + 1, 2*x - 4, domain=ZZ)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4, domain=QQ)
(x/2 + 1, 5)
```

`sympy.polys.polytools.rem(f, g, *gens, **args)`
Compute polynomial remainder of f and g .

Examples

```
>>> from sympy import rem, ZZ, QQ
>>> from sympy.abc import x
```

```
>>> rem(x**2 + 1, 2*x - 4, domain=ZZ)
x**2 + 1
>>> rem(x**2 + 1, 2*x - 4, domain=QQ)
5
```

`sympy.polys.polytools.quo(f, g, *gens, **args)`
Compute polynomial quotient of f and g .

Examples

```
>>> from sympy import quo
>>> from sympy.abc import x
```

```
>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1
```

`sympy.polys.polytools.exquo(f, g, *gens, **args)`
Compute polynomial exact quotient of f and g .

Examples

```
>>> from sympy import exquo
>>> from sympy.abc import x
```

```
>>> exquo(x**2 - 1, x - 1)
x + 1
```

```
>>> exquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.half_gcdex(f, g, *gens, **args)`
Half extended Euclidean algorithm of f and g .
Returns (s, h) such that $h = \gcd(f, g)$ and $s*f = h \pmod{g}$.

Examples

```
>>> from sympy import half_gcdex
>>> from sympy.abc import x
```

```
>>> half_gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(-x/5 + 3/5, x + 1)
```

`sympy.polys.polytools.gcdex(f, g, *gens, **args)`

Extended Euclidean algorithm of f and g .

Returns (s, t, h) such that $h = \gcd(f, g)$ and $s*f + t*g = h$.

Examples

```
>>> from sympy import gcdex
>>> from sympy.abc import x
```

```
>>> gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(-x/5 + 3/5, x**2/5 - 6*x/5 + 2, x + 1)
```

`sympy.polys.polytools.invert(f, g, *gens, **args)`

Invert f modulo g when possible.

See also:

`sympy.core.numbers.mod_inverse`

Examples

```
>>> from sympy import invert, S
>>> from sympy.core.numbers import mod_inverse
>>> from sympy.abc import x
```

```
>>> invert(x**2 - 1, 2*x - 1)
-4/3
```

```
>>> invert(x**2 - 1, x - 1)
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

For more efficient inversion of Rationals, use the `mod_inverse` function:

```
>>> mod_inverse(3, 5)
2
>>> (S(2)/5).invert(S(7)/3)
5/2
```

`sympy.polys.polytools.subresultants(f, g, *gens, **args)`

Compute subresultant PRS of f and g .

Examples

```
>>> from sympy import subresultants
>>> from sympy.abc import x
```

```
>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

`sympy.polys.polytools.resultant(f, g, *gens, **args)`
Compute resultant of f and g .

Examples

```
>>> from sympy import resultant
>>> from sympy.abc import x
```

```
>>> resultant(x**2 + 1, x**2 - 1)
4
```

`sympy.polys.polytools.discriminant(f, *gens, **args)`
Compute discriminant of f .

Examples

```
>>> from sympy import discriminant
>>> from sympy.abc import x
```

```
>>> discriminant(x**2 + 2*x + 3)
-8
```

`sympy.polys.dispersion.dispersion(p, q=None, *gens, **args)`
Compute the dispersion of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\text{dis}(f, g)$ is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x + a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial $\text{dis}(f) := \text{dis}(f, f)$. Note that we make the definition $\max\{\} := -\infty$.

See also:

[`dispersionset`](#) (page 809)

References

1. [ManWright94] (page 1867)
2. [Koepf98] (page 1867)
3. [Abramov71] (page 1867)

4. [Man93] (page 1867)

[ManWright94] (page 1867), [Koepf98] (page 1867), [Abramov71] (page 1867), [Man93] (page 1867)

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be $-\infty$ as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`sympy.polys.dispersion.dispersionset(p, q=None, *gens, **args)`

Compute the dispersion set of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is

defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

See also:

[dispersion](#) (page 808)

References

1. [\[ManWright94\]](#) (page 1867)
2. [\[Koepf98\]](#) (page 1867)
3. [\[Abramov71\]](#) (page 1867)
4. [\[Man93\]](#) (page 1867)

[\[ManWright94\]](#) (page 1867), [\[Koepf98\]](#) (page 1867), [\[Abramov71\]](#) (page 1867), [\[Man93\]](#) (page 1867)

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
```

(continues on next page)

(continued from previous page)

```
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`sympy.polys.polytools.terms_gcd(f, *gens, **args)`

Remove GCD of terms from `f`.

If the `deep` flag is `True`, then the arguments of `f` will have `terms_gcd` applied to them.

If a fraction is factored out of `f` and `f` is an `Add`, then an unevaluated `Mul` will be returned so that automatic simplification does not redistribute it. The hint `clear`, when set to `False`, can be used to prevent such factoring when all coefficients are not fractions.

See also:

[sympy.core.exprtools.gcd_terms](#) (page 200), [sympy.core.exprtools.factor_terms](#) (page 201)

Examples

```
>>> from sympy import terms_gcd, cos
>>> from sympy.abc import x, y
>>> terms_gcd(x**6*y**2 + x**3*y, x, y)
x**3*y*(x**3*y + 1)
```

The default action of polys routines is to expand the expression given to them. `terms_gcd` follows this behavior:

```
>>> terms_gcd((3+3*x)*(x+x*y))
3*x*(x*y + x + y + 1)
```

If this is not desired then the hint `expand` can be set to `False`. In this case the expression will be treated as though it were comprised of one or more terms:

```
>>> terms_gcd((3+3*x)*(x+x*y), expand=False)
(3*x + 3)*(x*y + x)
```

In order to traverse factors of a `Mul` or the arguments of other functions, the `deep` hint can be used:

```
>>> terms_gcd((3 + 3*x)*(x + x*y), expand=False, deep=True)
3*x*(x + 1)*(y + 1)
>>> terms_gcd(cos(x + x*y), deep=True)
cos(x*(y + 1))
```

Rationals are factored out by default:

```
>>> terms_gcd(x + y/2)
(2*x + y)/2
```

Only the y-term had a coefficient that was a fraction; if one does not want to factor out the 1/2 in cases like this, the flag `clear` can be set to `False`:

```
>>> terms_gcd(x + y/2, clear=False)
x + y/2
>>> terms_gcd(x*y/2 + y**2, clear=False)
y*(x/2 + y)
```

The `clear` flag is ignored if all coefficients are fractions:

```
>>> terms_gcd(x/3 + y/2, clear=False)
(2*x + 3*y)/6
```

`sympy.polys.polytools.cofactors(f, g, *gens, **args)`

Compute GCD and cofactors of `f` and `g`.

Returns polynomials `(h, cff, cfg)` such that `h = gcd(f, g)`, and `cff = quo(f, h)` and `cfg = quo(g, h)` are, so called, cofactors of `f` and `g`.

Examples

```
>>> from sympy import cofactors
>>> from sympy.abc import x
```

```
>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

`sympy.polys.polytools.gcd(f, g=None, *gens, **args)`

Compute GCD of `f` and `g`.

Examples

```
>>> from sympy import gcd
>>> from sympy.abc import x
```

```
>>> gcd(x**2 - 1, x**2 - 3*x + 2)
x - 1
```

`sympy.polys.polytools.gcd_list(seq, *gens, **args)`

Compute GCD of a list of polynomials.

Examples

```
>>> from sympy import gcd_list
>>> from sympy.abc import x
```

```
>>> gcd_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x - 1
```

`sympy.polys.polytools.lcm(f, g=None, *gens, **args)`

Compute LCM of `f` and `g`.

Examples

```
>>> from sympy import lcm
>>> from sympy.abc import x
```

```
>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

`sympy.polys.polytools.lcm_list(seq, *gens, **args)`
 Compute LCM of a list of polynomials.

Examples

```
>>> from sympy import lcm_list
>>> from sympy.abc import x
```

```
>>> lcm_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x**5 - x**4 - 2*x**3 - x**2 + x + 2
```

`sympy.polys.polytools.trunc(f, p, *gens, **args)`
 Reduce f modulo a constant p .

Examples

```
>>> from sympy import trunc
>>> from sympy.abc import x
```

```
>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1
```

`sympy.polys.polytools.monic(f, *gens, **args)`
 Divide all coefficients of f by $\text{LC}(f)$.

Examples

```
>>> from sympy import monic
>>> from sympy.abc import x
```

```
>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3
```

`sympy.polys.polytools.content(f, *gens, **args)`
 Compute GCD of coefficients of f .

Examples

```
>>> from sympy import content
>>> from sympy.abc import x
```

```
>>> content(6*x**2 + 8*x + 12)
2
```

`sympy.polys.polytools.primitive(f, *gens, **args)`
Compute content and the primitive form of f .

Examples

```
>>> from sympy.polys.polytools import primitive
>>> from sympy.abc import x
```

```
>>> primitive(6*x**2 + 8*x + 12)
(2, 3*x**2 + 4*x + 6)
```

```
>>> eq = (2 + 2*x)*x + 2
```

Expansion is performed by default:

```
>>> primitive(eq)
(2, x**2 + x + 1)
```

Set `expand` to `False` to shut this off. Note that the extraction will not be recursive; use the `as_content_primitive` method for recursive, non-destructive Rational extraction.

```
>>> primitive(eq, expand=False)
(1, x*(2*x + 2) + 2)
```

```
>>> eq.as_content_primitive()
(2, x*(x + 1) + 1)
```

`sympy.polys.polytools.compose(f, g, *gens, **args)`
Compute functional composition $f(g)$.

Examples

```
>>> from sympy import compose
>>> from sympy.abc import x
```

```
>>> compose(x**2 + x, x - 1)
x**2 - x
```

`sympy.polys.polytools.decompose(f, *gens, **args)`
Compute functional decomposition of f .

Examples

```
>>> from sympy import decompose
>>> from sympy.abc import x
```

```
>>> decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]
```

`sympy.polys.polytools.sturm(f, *gens, **args)`
 Compute Sturm sequence of f .

Examples

```
>>> from sympy import sturm
>>> from sympy.abc import x
```

```
>>> sturm(x**3 - 2*x**2 + x - 3)
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2*x/9 + 25/9, -2079/4]
```

`sympy.polys.polytools.gff_list(f, *gens, **args)`
 Compute a list of greatest factorial factors of f .

Note that the input to `ff()` and `rf()` should be Poly instances to use the definitions here.

Examples

```
>>> from sympy import gff_list, ff, Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**5 + 2*x**4 - x**3 - 2*x**2, x)
```

```
>>> gff_list(f)
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]
```

```
>>> (ff(Poly(x), 1)*ff(Poly(x + 2), 4)).expand() == f
True
```

```
>>> f = Poly(x**12 + 6*x**11 - 11*x**10 - 56*x**9 + 220*x**8 + 208*x**7 -
↪ 1401*x**6 + 1090*x**5 + 2715*x**4 - 6720*x**3 - 1092*x**2 + 5040*x, x)
```

```
>>> gff_list(f)
[(Poly(x**3 + 7, x, domain='ZZ'), 2), (Poly(x**2 + 5*x, x, domain='ZZ'), 3)]
```

```
>>> ff(Poly(x**3 + 7, x), 2)*ff(Poly(x**2 + 5*x, x), 3) == f
True
```

`sympy.polys.polytools.gff(f, *gens, **args)`
 Compute greatest factorial factorization of f .

`sympy.polys.polytools.sqf_norm(f, *gens, **args)`
 Compute square-free norm of f .

Returns s, f, r , such that $g(x) = f(x-sa)$ and $r(x) = \text{Norm}(g(x))$ is a square-free polynomial over K , where a is the algebraic extension of the ground domain.

Examples

```
>>> from sympy import sqf_norm, sqrt
>>> from sympy.abc import x
```

```
>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)
```

`sympy.polys.polytools.sqf_part(f, *gens, **args)`
Compute square-free part of f .

Examples

```
>>> from sympy import sqf_part
>>> from sympy.abc import x
```

```
>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2
```

`sympy.polys.polytools.sqf_list(f, *gens, **args)`
Compute a list of square-free factors of f .

Examples

```
>>> from sympy import sqf_list
>>> from sympy.abc import x
```

```
>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])
```

`sympy.polys.polytools.sqf(f, *gens, **args)`
Compute square-free factorization of f .

Examples

```
>>> from sympy import sqf
>>> from sympy.abc import x
```

```
>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

`sympy.polys.polytools.factor_list(f, *gens, **args)`
Compute a list of irreducible factors of f .

Examples

```
>>> from sympy import factor_list
>>> from sympy.abc import x, y
```

```
>>> factor_list(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
(2, [(x + y, 1), (x**2 + 1, 2)])
```

`sympy.polys.polytools.factor(f, *gens, **args)`

Compute the factorization of expression, `f`, into irreducibles. (To factor an integer into primes, use `factorint`.)

There two modes implemented: symbolic and formal. If `f` is not an instance of `Poly` (page 821) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, `factor()` (page 817) will traverse the expression tree and factor its components without any prior expansion, unless an instance of `Add` is encountered (in this case formal factorization is used). This way `factor()` (page 817) can handle large or symbolic exponents.

By default, the factorization is computed over the rationals. To factor over other domain, e.g. an algebraic or finite field, use appropriate options: `extension`, `modulus` or `domain`.

See also:

`sympy.ntheory.factor_.factorint` (page 312)

Examples

```
>>> from sympy import factor, sqrt
>>> from sympy.abc import x, y
```

```
>>> factor(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
2*(x + y)*(x**2 + 1)**2
```

```
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, gaussian=True)
(x - I)*(x + I)
```

```
>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
```

```
>>> factor((x**2 - 1)/(x**2 + 4*x + 4))
(x - 1)*(x + 1)/(x + 2)**2
>>> factor((x**2 + 4*x + 4)**10000000*(x**2 + 1))
(x + 2)**20000000*(x**2 + 1)
```

By default, `factor` deals with an expression as a whole:

```
>>> eq = 2*(x**2 + 2*x + 1)
>>> factor(eq)
2*(x**2 + 2*x + 1)
```

If the `deep` flag is `True` then subexpressions will be factored:

```
>>> factor(eq, deep=True)
2**((x + 1)**2)
```

`sympy.polys.polytools.intervals`(*F*, *all*=False, *eps*=None, *inf*=None, *sup*=None, *strict*=False, *fast*=False, *sqf*=False)
Compute isolating intervals for roots of *f*.

Examples

```
>>> from sympy import intervals
>>> from sympy.abc import x
```

```
>>> intervals(x**2 - 3)
[((-2, -1), 1), ((1, 2), 1)]
>>> intervals(x**2 - 3, eps=1e-2)
[(-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

`sympy.polys.polytools.refine_root`(*f*, *s*, *t*, *eps*=None, *steps*=None, *fast*=False, *check_sqf*=False)
Refine an isolating interval of a root to the given precision.

Examples

```
>>> from sympy import refine_root
>>> from sympy.abc import x
```

```
>>> refine_root(x**2 - 3, 1, 2, eps=1e-2)
(19/11, 26/15)
```

`sympy.polys.polytools.count_roots`(*f*, *inf*=None, *sup*=None)
Return the number of roots of *f* in [*inf*, *sup*] interval.
If one of *inf* or *sup* is complex, it will return the number of roots in the complex rectangle with corners at *inf* and *sup*.

Examples

```
>>> from sympy import count_roots, I
>>> from sympy.abc import x
```

```
>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1
```

`sympy.polys.polytools.real_roots`(*f*, *multiple*=True)
Return a list of real roots with multiplicities of *f*.

Examples

```
>>> from sympy import real_roots
>>> from sympy.abc import x
```



```
>>> real_roots(2*x**3 - 7*x**2 + 4*x + 4)
[-1/2, 2, 2]
```

`sympy.polys.polytools.nroots(f, n=15, maxsteps=50, cleanup=True)`
 Compute numerical approximations of roots of `f`.

Examples

```
>>> from sympy import nroots
>>> from sympy.abc import x
```

```
>>> nroots(x**2 - 3, n=15)
[-1.73205080756888, 1.73205080756888]
>>> nroots(x**2 - 3, n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

`sympy.polys.polytools.ground_roots(f, *gens, **args)`
 Compute roots of `f` by factorization in the ground domain.

Examples

```
>>> from sympy import ground_roots
>>> from sympy.abc import x
```

```
>>> ground_roots(x**6 - 4*x**4 + 4*x**3 - x**2)
{0: 2, 1: 2}
```

`sympy.polys.polytools.nth_power_roots_poly(f, n, *gens, **args)`
 Construct a polynomial with `n`-th powers of roots of `f`.

Examples

```
>>> from sympy import nth_power_roots_poly, factor, roots
>>> from sympy.abc import x
```

```
>>> f = x**4 - x**2 + 1
>>> g = factor(nth_power_roots_poly(f, 2))
```

```
>>> g
(x**2 - x + 1)**2
```

```
>>> R_f = [ (r**2).expand() for r in roots(f) ]
>>> R_g = roots(g).keys()
```

```
>>> set(R_f) == set(R_g)
True
```

`sympy.polys.polytools.cancel(f, *gens, **args)`
 Cancel common factors in a rational function `f`.

Examples

```
>>> from sympy import cancel, sqrt, Symbol
>>> from sympy.abc import x
>>> A = Symbol('A', commutative=False)
```

```
>>> cancel((2*x**2 - 2)/(x**2 - 2*x + 1))
(2*x + 2)/(x - 1)
>>> cancel((sqrt(3) + sqrt(15)*A)/(sqrt(2) + sqrt(10)*A))
sqrt(6)/2
```

`sympy.polys.polytools.reduced(f, G, *gens, **args)`
Reduces a polynomial f modulo a set of polynomials G .

Given a polynomial f and a set of polynomials $G = (g_1, \dots, g_n)$, computes a set of quotients $q = (q_1, \dots, q_n)$ and the remainder r such that $f = q_1 g_1 + \dots + q_n g_n + r$, where r vanishes or r is a completely reduced polynomial with respect to G .

Examples

```
>>> from sympy import reduced
>>> from sympy.abc import x, y
```

```
>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

`sympy.polys.polytools.groebner(F, *gens, **args)`
Computes the reduced Groebner basis for a set of polynomials.

Use the `order` argument to set the monomial ordering that will be used to compute the basis. Allowed orders are `lex`, `grlex` and `grevlex`. If no order is specified, it defaults to `lex`.

For more information on Groebner bases, see the references and the docstring of `solve_poly_system()`.

References

1. [Buchberger01] (page 1866)
2. [Cox97] (page 1866)

[Buchberger01] (page 1866), [Cox97] (page 1866)

Examples

Example taken from [1].

```
>>> from sympy import groebner
>>> from sympy.abc import x, y
```

```
>>> F = [x*y - 2*y, 2*y**2 - x**2]
```

```
>>> groebner(F, x, y, order='lex')
GroebnerBasis([x**2 - 2*y**2, x*y - 2*y, y**3 - 2*y], x, y,
               domain='ZZ', order='lex')
>>> groebner(F, x, y, order='grlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
               domain='ZZ', order='grlex')
>>> groebner(F, x, y, order='grevlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
               domain='ZZ', order='grevlex')
```

By default, an improved implementation of the Buchberger algorithm is used. Optionally, an implementation of the F5B algorithm can be used. The algorithm can be set using method flag or with the `setup()` function from `sympy.polys.polyconfig`:

```
>>> F = [x**2 - x - 1, (2*x - 1) * y - (x**10 - (1 - x)**10)]
```

```
>>> groebner(F, x, y, method='buchberger')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
>>> groebner(F, x, y, method='f5b')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
```

`sympy.polys.polytools.is_zero_dimensional(F, *gens, **args)`

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of `F` is bounded.

References

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

class `sympy.polys.polytools.Poly`

Generic class for representing and operating on polynomial expressions. Subclasses `Expr` class.

See also:

`sympy.core.expr.Expr` (page 111)

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

Create a univariate polynomial:

```
>>> Poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

Create a univariate polynomial with specific domain:

```
>>> from sympy import sqrt
>>> Poly(x**2 + 2*x + sqrt(3), domain='R')
Poly(1.0*x**2 + 2.0*x + 1.73205080756888, x, domain='RR')
```

Create a multivariate polynomial:

```
>>> Poly(y*x**2 + x*y + 1)
Poly(x**2*y + x*y + 1, x, y, domain='ZZ')
```

Create a univariate polynomial, where y is a constant:

```
>>> Poly(y*x**2 + x*y + 1, x)
Poly(y*x**2 + y*x + 1, x, domain='ZZ[y]')
```

You can evaluate the above polynomial as a function of y:

```
>>> Poly(y*x**2 + x*y + 1, x).eval(2)
6*y + 1
```

EC(order=None)

Returns the last non-zero coefficient of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).EC()
3
```

EM(order=None)

Returns the last non-zero monomial of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).EM()
x**0*y**1
```

ET(order=None)

Returns the last non-zero term of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).ET()
(x**0*y**1, 3)
```

LC(order=None)
Returns the leading coefficient of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(4*x**3 + 2*x**2 + 3*x, x).LC()
4
```

LM(order=None)
Returns the leading monomial of f.
The Leading monomial signifies the monomial having the highest power of the principal generator in the expression f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LM()
x**2*y**0
```

LT(order=None)
Returns the leading term of f.
The Leading term signifies the term having the highest power of the principal generator in the expression f along with its coefficient.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LT()
(x**2*y**0, 4)
```

TC()
Returns the trailing coefficient of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).TC()
0
```

abs()

Make all coefficients in f positive.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).abs()
Poly(x**2 + 1, x, domain='ZZ')
```

add(g)

Add two polynomials f and g .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).add(Poly(x - 2, x))
Poly(x**2 + x - 1, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) + Poly(x - 2, x)
Poly(x**2 + x - 1, x, domain='ZZ')
```

add_ground(coeff)

Add an element of the ground domain to f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).add_ground(2)
Poly(x + 3, x, domain='ZZ')
```

all_coeffs()

Returns all coefficients from a univariate polynomial f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_coeffs()
[1, 0, 2, -1]
```

all_monoms()

Returns all monomials from a univariate polynomial f .

See also:

[all_terms](#) (page 825)

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_monoms()
[(3,), (2,), (1,), (0,)]
```

all_roots(multiple=True, radicals=True)

Return a list of real and complex roots with multiplicities.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).all_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).all_roots()
[CRootOf(x**3 + x + 1, 0),
 CRootOf(x**3 + x + 1, 1),
 CRootOf(x**3 + x + 1, 2)]
```

all_terms()

Returns all terms from a univariate polynomial f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_terms()
[((3,), 1), ((2,), 0), ((1,), 2), ((0,), -1)]
```

args

Don't mess up with the core.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).args
(x**2 + 1,)
```

as_dict(native=False, zero=False)
Switch to a dict representation.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 - y, x, y).as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}
```

as_expr(*gens)
Convert a Poly instance to an Expr instance.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2 + 2*x*y**2 - y, x, y)
```

```
>>> f.as_expr()
x**2 + 2*x*y**2 - y
>>> f.as_expr({x: 5})
10*y**2 - y + 25
>>> f.as_expr(5, 6)
379
```

as_list(native=False)
Switch to a list representation.

cancel(g, include=False)
Cancel common factors in a rational function f/g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x))
(1, Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

```
>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x), include=True)
(Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

clear_denoms(convert=False)
Clear denominators, but keep the ground domain.

Examples

```
>>> from sympy import Poly, S, QQ
>>> from sympy.abc import x
```

```
>>> f = Poly(x/2 + S(1)/3, x, domain=QQ)
```

```
>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))
>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

coeff_monomial(monom)

Returns the coefficient of monom in *f* if there, else None.

See also:

nth (page 848) more efficient query using exponents of the monomial's generators

Examples

```
>>> from sympy import Poly, exp
>>> from sympy.abc import x, y
```

```
>>> p = Poly(24*x*y*exp(8) + 23*x, x, y)
```

```
>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*exp(8)
```

Note that `Expr.coeff()` behaves differently, collecting terms if possible; the `Poly` must be converted to an `Expr` to use that method, however:

```
>>> p.as_expr().coeff(x)
24*y*exp(8) + 23
>>> p.as_expr().coeff(y)
24*x*exp(8)
>>> p.as_expr().coeff(x*y)
24*exp(8)
```

coeffs(order=None)

Returns all non-zero coefficients from *f* in lex order.

See also:

all_coeffs (page 824), **coeff_monomial** (page 827), **nth** (page 848)

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

cofactors(g)

Returns the GCD of f and g and their cofactors.

Returns polynomials (h, cff, cfg) such that $h = \gcd(f, g)$, and $cff = \text{quo}(f, h)$ and $cfg = \text{quo}(g, h)$ are, so called, cofactors of f and g .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).cofactors(Poly(x**2 - 3*x + 2, x))
(Poly(x - 1, x, domain='ZZ'),
 Poly(x + 1, x, domain='ZZ'),
 Poly(x - 2, x, domain='ZZ'))
```

compose(g)

Computes the functional composition of f and g .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + x, x).compose(Poly(x - 1, x))
Poly(x**2 - x, x, domain='ZZ')
```

content()

Returns the GCD of polynomial coefficients.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(6*x**2 + 8*x + 12, x).content()
2
```

count_roots(inf=None, sup=None)

Return the number of roots of f in $[inf, sup]$ interval.

Examples

```
>>> from sympy import Poly, I
>>> from sympy.abc import x
```

```
>>> Poly(x**4 - 4, x).count_roots(-3, 3)
2
>>> Poly(x**4 - 4, x).count_roots(0, 1 + 3*I)
1
```

decompose()

Computes a functional decomposition of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**4 + 2*x**3 - x - 1, x, domain='ZZ').decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]
```

deflate()

Reduce degree of f by mapping x_i^{**m} to y_i .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3 + 1, x, y).deflate()
((3, 2), Poly(x**2*y + x + 1, x, y, domain='ZZ'))
```

degree(gen=0)

Returns degree of f in x_j .

The degree of 0 is negative infinity.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).degree()
2
>>> Poly(x**2 + y*x + y, x, y).degree(y)
1
>>> Poly(0, x).degree()
-oo
```

degree_list()

Returns a list of degrees of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).degree_list()
(2, 1)
```

diff(*specs, **kwargs)
Computes partial derivative of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).diff()
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x*y**2 + x, x, y).diff((0, 0), (1, 1))
Poly(2*x*y, x, y, domain='ZZ')
```

discriminant()
Computes the discriminant of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

dispersion(g=None)
Compute the dispersion of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\text{dis}(f, g)$ is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial $\text{dis}(f) := \text{dis}(f, f)$.

See also:

[dispersionset](#) (page 831)

References

1. [\[ManWright94\]](#) (page 1867)

2. [Koepf98] (page 1867)
3. [Abramov71] (page 1867)
4. [Man93] (page 1867)

[ManWright94] (page 1867), [Koepf98] (page 1867), [Abramov71] (page 1867), [Man93] (page 1867)

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

dispersionset(g=None)

Compute the dispersion set of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

See also:

[dispersion](#) (page 830)

References

1. [\[ManWright94\]](#) (page 1867)
2. [\[Koepf98\]](#) (page 1867)
3. [\[Abramov71\]](#) (page 1867)
4. [\[Man93\]](#) (page 1867)

[\[ManWright94\]](#) (page 1867), [\[Koepf98\]](#) (page 1867), [\[Abramov71\]](#) (page 1867), [\[Man93\]](#) (page 1867)

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

div(g, auto=True)

Polynomial division with remainder of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x))
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

domain

Get the ground domain of self.

eject(*gens)

Eject selected generators into the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x, y)
```

```
>>> f.eject(x)
Poly(x*y**3 + (x**2 + x)*y + 1, y, domain='ZZ[x]')
>>> f.eject(y)
Poly(y*x**2 + (y**3 + y)*x + 1, x, domain='ZZ[y]')
```

eval(x, a=None, auto=True)

Evaluate f at a in the given variable.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x**2 + 2*x + 3, x).eval(2)
11
```

```
>>> Poly(2*x*y + 3*x + y + 2, x, y).eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')
```

```
>>> f = Poly(2*x*y + 3*x + y + 2*z, x, y, z)
```

```
>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45
```

```
>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')
>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')
```

`exclude()`

Remove unnecessary generators from `f`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import a, b, c, d, x
```

```
>>> Poly(a + x, a, b, c, d, x).exclude()
Poly(a + x, a, x, domain='ZZ')
```

`exquo(g, auto=True)`

Computes polynomial exact quotient of `f` by `g`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).exquo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).exquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```


exquo_ground(coeff)

Exact quotient of f by a an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x + 4).exquo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).exquo_ground(2)
Traceback (most recent call last):
...
ExactQuotientFailed: 2 does not divide 3 in ZZ
```

factor_list()

Returns a list of irreducible factors of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

```
>>> Poly(f).factor_list()
(2, [(Poly(x + y, x, y, domain='ZZ'), 1),
      (Poly(x**2 + 1, x, y, domain='ZZ'), 2)])
```

factor_list_include()

Returns a list of irreducible factors of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

```
>>> Poly(f).factor_list_include()
[(Poly(2*x + 2*y, x, y, domain='ZZ'), 1),
 (Poly(x**2 + 1, x, y, domain='ZZ'), 2)]
```

free_symbols

Free symbols of a polynomial expression.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x**2 + 1).free_symbols
{x}
>>> Poly(x**2 + y).free_symbols
{x, y}
>>> Poly(x**2 + y, x).free_symbols
{x, y}
>>> Poly(x**2 + y, x, z).free_symbols
{x, y}
```

free_symbols_in_domain

Free symbols of the domain of self.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 1).free_symbols_in_domain
set()
>>> Poly(x**2 + y).free_symbols_in_domain
set()
>>> Poly(x**2 + y, x).free_symbols_in_domain
{y}
```

classmethod from_dict(rep, *gens, **args)

Construct a polynomial from a dict.

classmethod from_expr(rep, *gens, **args)

Construct a polynomial from an expression.

classmethod from_list(rep, *gens, **args)

Construct a polynomial from a list.

classmethod from_poly(rep, *gens, **args)

Construct a polynomial from a polynomial.

gcd(g)

Returns the polynomial GCD of f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).gcd(Poly(x**2 - 3*x + 2, x))
Poly(x - 1, x, domain='ZZ')
```

gcdex(g, auto=True)

Extended Euclidean algorithm of f and g.

Returns (s, t, h) such that $h = \gcd(f, g)$ and $s*f + t*g = h$.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'),
 Poly(1/5*x**2 - 6/5*x + 2, x, domain='QQ'),
 Poly(x + 1, x, domain='QQ'))
```

gen

Return the principal generator.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).gen
x
```

get_domain()

Get the ground domain of f .

get_modulus()

Get the modulus of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, modulus=2).get_modulus()
2
```

gff_list()

Computes greatest factorial factorization of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**5 + 2*x**4 - x**3 - 2*x**2
```

```
>>> Poly(f).gff_list()
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]
```

ground_roots()

Compute roots of f by factorization in the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**6 - 4*x**4 + 4*x**3 - x**2).ground_roots()
{0: 2, 1: 2}
```

half_gcdex(g, auto=True)

Half extended Euclidean algorithm of f and g .

Returns (s, h) such that $h = \gcd(f, g)$ and $s*f = h \pmod{g}$.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).half_gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```

has_only_gens(*gens)

Return True if $\text{Poly}(f, *gens)$ retains ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x*y + 1, x, y, z).has_only_gens(x, y)
True
>>> Poly(x*y + z, x, y, z).has_only_gens(x, y)
False
```

homogeneous_order()

Returns the homogeneous order of f .

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. This degree is the homogeneous order of

f. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 841).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**5 + 2*x**3*y**2 + 9*x*y**4)
>>> f.homogeneous_order()
5
```

homogenize(s)

Returns the homogeneous polynomial of f.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 841). If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 838).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> f = Poly(x**5 + 2*x**2*y**2 + 9*x*y**3)
>>> f.homogenize(z)
Poly(x**5 + 2*x**2*y**2*z + 9*x*y**3*z, x, y, z, domain='ZZ')
```

inject(front=False)

Inject ground domain generators into f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x)
```

```
>>> f.inject()
Poly(x**2*y + x*y**3 + x*y + 1, x, y, domain='ZZ')
>>> f.inject(front=True)
Poly(y**3*x + y*x**2 + y*x + 1, y, x, domain='ZZ')
```

integrate(*specs, **args)

Computes indefinite integral of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')
```

```
>>> Poly(x*y**2 + x, x, y).integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

intervals(all=False, eps=None, inf=None, sup=None, fast=False, sqf=False)

Compute isolating intervals for roots of f .

For real roots the Vincent-Akritas-Strzebonski (VAS) continued fractions method is used.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3, x).intervals()
[((-2, -1), 1), ((1, 2), 1)]
>>> Poly(x**2 - 3, x).intervals(eps=1e-2)
[((-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

References:

1. Alkiviadis G. Akritas and Adam W. Strzebonski: A Comparative Study of Two Real Root Isolation Methods . Nonlinear Analysis: Modelling and Control, Vol. 10, No. 4, 297-304, 2005. 2. Alkiviadis G. Akritas, Adam W. Strzebonski and Panagiotis S. Vigklas: Improving the Performance of the Continued Fractions Method Using new Bounds of Positive Roots. Nonlinear Analysis: Modelling and Control, Vol. 13, No. 3, 265-279, 2008.

invert(g, auto=True)

Invert f modulo g when possible.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).invert(Poly(2*x - 1, x))
Poly(-4/3, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).invert(Poly(x - 1, x))
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

is_cyclotomic

Returns True if f is a cyclotomic polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
```

```
>>> Poly(f).is_cyclotomic
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
```

```
>>> Poly(g).is_cyclotomic
True
```

is_ground

Returns True if f is an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x, x).is_ground
False
>>> Poly(2, x).is_ground
True
>>> Poly(y, x).is_ground
True
```

is_homogeneous

Returns True if f is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 838).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x*y, x, y).is_homogeneous
True
>>> Poly(x**3 + x*y, x, y).is_homogeneous
False
```

is_irreducible

Returns True if f has no factors over its domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + x + 1, x, modulus=2).is_irreducible
True
>>> Poly(x**2 + 1, x, modulus=2).is_irreducible
False
```

is_linear

Returns True if f is linear in all its variables.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x + y + 2, x, y).is_linear
True
>>> Poly(x*y + 2, x, y).is_linear
False
```

is_monic

Returns True if the leading coefficient of f is one.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 2, x).is_monic
True
>>> Poly(2*x + 2, x).is_monic
False
```

is_monomial

Returns True if f is zero or has only one term.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(3*x**2, x).is_monomial
True
>>> Poly(3*x**2 + 1, x).is_monomial
False
```

is_multivariate

Returns True if f is a multivariate polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_multivariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_multivariate
True
>>> Poly(x*y**2 + x*y + 1, x).is_multivariate
False
>>> Poly(x**2 + x + 1, x, y).is_multivariate
True
```

is_one

Returns True if f is a unit polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(0, x).is_one
False
>>> Poly(1, x).is_one
True
```

is_primitive

Returns True if GCD of the coefficients of f is one.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 + 6*x + 12, x).is_primitive
False
>>> Poly(x**2 + 3*x + 6, x).is_primitive
True
```

is_quadratic

Returns True if f is quadratic in all its variables.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x*y + 2, x, y).is_quadratic
True
>>> Poly(x*y**2 + 2, x, y).is_quadratic
False
```

is_sqf

Returns True if f is a square-free polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).is_sqf
False
>>> Poly(x**2 - 1, x).is_sqf
True
```

is_univariate

Returns True if f is a univariate polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_univariate
True
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate
False
>>> Poly(x*y**2 + x*y + 1, x).is_univariate
True
>>> Poly(x**2 + x + 1, x, y).is_univariate
False
```

is_zero

Returns True if f is a zero polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(0, x).is_zero
True
>>> Poly(1, x).is_zero
False
```

l1_norm()
Returns l1 norm of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).l1_norm()
6
```

lcm(g)
Returns polynomial LCM of f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).lcm(Poly(x**2 - 3*x + 2, x))
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')
```

length()
Returns the number of non-zero terms in f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 2*x - 1).length()
3
```

lift()
Convert algebraic coefficients to rationals.

Examples

```
>>> from sympy import Poly, I
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + I*x + 1, x, extension=I).lift()
Poly(x**4 + 3*x**2 + 1, x, domain='QQ')
```

ltrim(gen)

Remove dummy generators from f that are to the left of specified gen in the generators as ordered. When gen is an integer, it refers to the generator located at that position within the tuple of generators of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(y**2 + y*z**2, x, y, z).ltrim(y)
Poly(y**2 + y*z**2, y, z, domain='ZZ')
>>> Poly(z, x, y, z).ltrim(-1)
Poly(z, z, domain='ZZ')
```

max_norm()

Returns maximum norm of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).max_norm()
3
```

monic(auto=True)

Divides all coefficients by $LC(f)$.

Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x
```

```
>>> Poly(3*x**2 + 6*x + 9, x, domain=ZZ).monic()
Poly(x**2 + 2*x + 3, x, domain='QQ')
```

```
>>> Poly(3*x**2 + 4*x + 2, x, domain=ZZ).monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')
```

monoms(order=None)

Returns all non-zero monomials from f in lex order.

See also:

[all_monoms](#) (page 824)

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]
```

mul(g)

Multiply two polynomials f and g .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).mul(Poly(x - 2, x))
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x)*Poly(x - 2, x)
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

mul_ground(coeff)

Multiply f by a an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).mul_ground(2)
Poly(2*x + 2, x, domain='ZZ')
```

neg()

Negate all coefficients in f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).neg()
Poly(-x**2 + 1, x, domain='ZZ')
```

```
>>> -Poly(x**2 - 1, x)
Poly(-x**2 + 1, x, domain='ZZ')
```

classmethod `new(rep, *gens)`

Construct `Poly` (page 821) instance from raw representation.

norm()

Computes the product, $\text{Norm}(f)$, of the conjugates of a polynomial f defined over a number field K .

Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x
```

```
>>> a, b = sqrt(2), sqrt(3)
```

A polynomial over a quadratic extension. Two conjugates $x - a$ and $x + a$.

```
>>> f = Poly(x - a, x, extension=a)
>>> f.norm()
Poly(x**2 - 2, x, domain='QQ')
```

A polynomial over a quartic extension. Four conjugates $x - a$, $x - a$, $x + a$ and $x + a$.

```
>>> f = Poly(x - a, x, extension=(a, b))
>>> f.norm()
Poly(x**4 - 4*x**2 + 4, x, domain='QQ')
```

roots($n=15$, $\text{maxsteps}=50$, $\text{cleanup}=\text{True}$)

Compute numerical approximations of roots of f .

Parameters n ... the number of digits to calculate

maxsteps ... the maximum number of iterations to do

If the accuracy ' n ' cannot be reached in ' maxsteps ', it will raise an exception. You need to rerun with higher maxsteps .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3).nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> Poly(x**2 - 3).nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

nth(*N)

Returns the n-th coefficient of f where N are the exponents of the generators in the term of interest.

See also:

[coeff_monomial](#) (page 827)

Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x, y

>>> Poly(x**3 + 2*x**2 + 3*x, x).nth(2)
2
>>> Poly(x**3 + 2*x*y**2 + y**2, x, y).nth(1, 2)
2
>>> Poly(4*sqrt(x)*y)
Poly(4*y*(sqrt(x)), y, sqrt(x), domain='ZZ')
>>> _.nth(1, 1)
4
```

nth_power_roots_poly(n)

Construct a polynomial with n-th powers of roots of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = Poly(x**4 - x**2 + 1)

>>> f.nth_power_roots_poly(2)
Poly(x**4 - 2*x**3 + 3*x**2 - 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(3)
Poly(x**4 + 2*x**2 + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(4)
Poly(x**4 + 2*x**3 + 3*x**2 + 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(12)
Poly(x**4 - 4*x**3 + 6*x**2 - 4*x + 1, x, domain='ZZ')
```

one

Return one polynomial with self's properties.

pdiv(g)

Polynomial pseudo-division of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).pdiv(Poly(2*x - 4, x))
(Poly(2*x + 4, x, domain='ZZ'), Poly(20, x, domain='ZZ'))
```

per(rep, gens=None, remove=None)
Create a Poly out of the given representation.

Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x, y
```

```
>>> from sympy.polys.polyclasses import DMP
```

```
>>> a = Poly(x**2 + 1)
```

```
>>> a.per(DMP([ZZ(1), ZZ(1)], ZZ), gens=[y])
Poly(y + 1, y, domain='ZZ')
```

pexquo(g)
Polynomial exact pseudo-quotient of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).pexquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).pexquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

pow(n)
Raise f to a non-negative power n.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x - 2, x).pow(3)
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**3
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```


pquo(g)

Polynomial pseudo-quotient of f by g .

See the Caveat note in the function `prem(f, g)`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).pquo(Poly(2*x - 4, x))
Poly(2*x + 4, x, domain='ZZ')
```

```
>>> Poly(x**2 - 1, x).pquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

prem(g)

Polynomial pseudo-remainder of f by g .

Caveat: The function `prem(f, g, x)` can be safely used to compute in $\mathbb{Z}[x]$ only subresultant polynomial remainder sequences (prs's).

To safely compute Euclidean and Sturmian prs's in $\mathbb{Z}[x]$ employ anyone of the corresponding functions found in the module `sympy.polys.subresultants_qq_zz`. The functions in the module with suffix `_pg` compute prs's in $\mathbb{Z}[x]$ employing `rem(f, g, x)`, whereas the functions with suffix `_amv` compute prs's in $\mathbb{Z}[x]$ employing `rem_z(f, g, x)`.

The function `rem_z(f, g, x)` differs from `prem(f, g, x)` in that to compute the remainder polynomials in $\mathbb{Z}[x]$ it premultiplies the dividend times the absolute value of the leading coefficient of the divisor raised to the power `degree(f, x) - degree(g, x) + 1`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).prem(Poly(2*x - 4, x))
Poly(20, x, domain='ZZ')
```

primitive()

Returns the content and a primitive form of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 + 8*x + 12, x).primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

quo(g, auto=True)
Computes polynomial quotient of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).quo(Poly(2*x - 4, x))
Poly(1/2*x + 1, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).quo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

quo_ground(coeff)
Quotient of f by a an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x + 4).quo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).quo_ground(2)
Poly(x + 1, x, domain='ZZ')
```

rat_clear_denoms(g)
Clear denominators in a rational function f/g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2/y + 1, x)
>>> g = Poly(x**3 + y, x)
```

```
>>> p, q = f.rat_clear_denoms(g)
```

```
>>> p
Poly(x**2 + y, x, domain='ZZ[y]')
>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')
```

real_roots(multiple=True, radicals=True)
Return a list of real roots with multiplicities.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).real_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).real_roots()
[CRootOf(x**3 + x + 1, 0)]
```

refine_root(s, t, eps=None, steps=None, fast=False, check_sqf=False)
 Refine an isolating interval of a root to the given precision.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

rem(g, auto=True)
 Computes the polynomial remainder of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x))
Poly(5, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x), auto=False)
Poly(x**2 + 1, x, domain='ZZ')
```

reorder(*gens, **args)
 Efficiently apply new order of generators.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x*y**2, x, y).reorder(y, x)
Poly(y**2*x + x**2, y, x, domain='ZZ')
```

replace(x, y=None)
 Replace x with y in generators list.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 1, x).replace(x, y)
Poly(y**2 + 1, y, domain='ZZ')
```

resultant(g, includePRS=False)

Computes the resultant of f and g via PRS.

If includePRS=True, it includes the subresultant PRS in the result. Because the PRS is used to calculate the resultant, this is more efficient than calling `subresultants()` (page 807) separately.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**2 + 1, x)
```

```
>>> f.resultant(Poly(x**2 - 1, x))
4
>>> f.resultant(Poly(x**2 - 1, x), includePRS=True)
(4, [Poly(x**2 + 1, x, domain='ZZ'), Poly(x**2 - 1, x, domain='ZZ'),
     Poly(-2, x, domain='ZZ')])
```

retract(field=None)

Recalculate the ground domain of a polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**2 + 1, x, domain='QQ[y]')
>>> f
Poly(x**2 + 1, x, domain='QQ[y]')
```

```
>>> f.retract()
Poly(x**2 + 1, x, domain='ZZ')
>>> f.retract(field=True)
Poly(x**2 + 1, x, domain='QQ')
```

revert(n)

Compute $f^{**}(-1) \bmod x^{**n}$.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(1, x).revert(2)
Poly(1, x, domain='ZZ')
```

```
>>> Poly(1 + x, x).revert(1)
Poly(1, x, domain='ZZ')
```

```
>>> Poly(x**2 - 1, x).revert(1)
Traceback (most recent call last):
...
NotReversible: only unity is reversible in a ring
```

```
>>> Poly(1/x, x).revert(1)
Traceback (most recent call last):
...
PolynomialError: 1/x contains an element of the generators set
```

root(index, radicals=True)
Get an indexed root of a polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(2*x**3 - 7*x**2 + 4*x + 4)
```

```
>>> f.root(0)
-1/2
>>> f.root(1)
2
>>> f.root(2)
2
>>> f.root(3)
Traceback (most recent call last):
...
IndexError: root index out of [-3, 2] range, got 3
```

```
>>> Poly(x**5 + x + 1).root(0)
CRootOf(x**3 - x**2 + 1, 0)
```

set_domain(domain)
Set the ground domain of f.

set_modulus(modulus)
Set the modulus of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(5*x**2 + 2*x - 1, x).set_modulus(2)
Poly(x**2 + 1, x, modulus=2)
```

shift(a)

Efficiently compute Taylor shift $f(x + a)$.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')
```

slice(x, m, n=None)

Take a continuous subsequence of terms of f .

sqf_list(all=False)

Returns a list of square-free factors of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = 2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16
```

```
>>> Poly(f).sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])
```

```
>>> Poly(f).sqf_list(all=True)
(2, [(Poly(1, x, domain='ZZ'), 1),
      (Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])
```

sqf_list_include(all=False)

Returns a list of square-free factors of f .

Examples

```
>>> from sympy import Poly, expand
>>> from sympy.abc import x
```

```
>>> f = expand(2*(x + 1)**3*x**4)
>>> f
2*x**7 + 6*x**6 + 6*x**5 + 2*x**4
```

```
>>> Poly(f).sqf_list_include()
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

```
>>> Poly(f).sqf_list_include(all=True)
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(1, x, domain='ZZ'), 2),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

sqf_norm()

Computes square-free norm of f .

Returns s, f, r , such that $g(x) = f(x-sa)$ and $r(x) = \text{Norm}(g(x))$ is a square-free polynomial over K , where a is the algebraic extension of the ground domain.

Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x
```

```
>>> s, f, r = Poly(x**2 + 1, x, extension=[sqrt(3)]).sqf_norm()
```

```
>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

sqf_part()

Computes square-free part of f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 - 3*x - 2, x).sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

sqr()

Square a polynomial f .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x - 2, x).sqr()
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**2
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

sturm(auto=True)
Computes the Sturm sequence of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 - 2*x**2 + x - 3, x).sturm()
[Poly(x**3 - 2*x**2 + x - 3, x, domain='QQ'),
 Poly(3*x**2 - 4*x + 1, x, domain='QQ'),
 Poly(2/9*x + 25/9, x, domain='QQ'),
 Poly(-2079/4, x, domain='QQ')]
```

sub(g)
Subtract two polynomials f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).sub(Poly(x - 2, x))
Poly(x**2 - x + 3, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) - Poly(x - 2, x)
Poly(x**2 - x + 3, x, domain='ZZ')
```

sub_ground(coeff)
Subtract an element of the ground domain from f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).sub_ground(2)
Poly(x - 1, x, domain='ZZ')
```


subresultants(g)

Computes the subresultant PRS of f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).subresultants(Poly(x**2 - 1, x))
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

terms(order=None)

Returns all non-zero terms from f in lex order.

See also:

[all_terms](#) (page 825)

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).terms()
[(2, 0), (1, 2), (1, 1), (0, 1), (0, 3)]
```

terms_gcd()

Remove GCD of terms from the polynomial f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3*y, x, y).terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

termwise(func, *gens, **args)

Apply a function to all terms of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> def func(k, coeff):
...     k = k[0]
...     return coeff//10**(2-k)
```

```
>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

to_exact()

Make the ground domain exact.

Examples

```
>>> from sympy import Poly, RR
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1.0, x, domain=RR).to_exact()
Poly(x**2 + 1, x, domain='QQ')
```

to_field()

Make the ground domain a field.

Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x, domain=ZZ).to_field()
Poly(x**2 + 1, x, domain='QQ')
```

to_ring()

Make the ground domain a ring.

Examples

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, domain=QQ).to_ring()
Poly(x**2 + 1, x, domain='ZZ')
```

total_degree()

Returns the total degree of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).total_degree()
2
>>> Poly(x + y**5, x, y).total_degree()
5
```

transform(p, q)

Efficiently evaluate the functional transformation $q^{**n} * f(p/q)$.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).transform(Poly(x + 1, x), Poly(x - 1, x))
Poly(4, x, domain='ZZ')
```

trunc(p)

Reduce f modulo a constant p .

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 + 3*x**2 + 5*x + 7, x).trunc(3)
Poly(-x**3 - x + 1, x, domain='ZZ')
```

unify(g)

Make f and g belong to the same domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f, g = Poly(x/2 + 1, x), Poly(2*x + 1)
```

```
>>> f
Poly(1/2*x + 1, x, domain='QQ')
>>> g
Poly(2*x + 1, x, domain='ZZ')
```

```
>>> F, G = f.unify(g)
```

```
>>> F
Poly(1/2*x + 1, x, domain='QQ')
>>> G
Poly(2*x + 1, x, domain='QQ')
```

unit

Return unit polynomial with `self`'s properties.

zero

Return zero polynomial with `self`'s properties.

class `sympy.polys.polytools.PurePoly`
Class for representing pure polynomials.

free_symbols
Free symbols of a polynomial.

Examples

```
>>> from sympy import PurePoly
>>> from sympy.abc import x, y
```

```
>>> PurePoly(x**2 + 1).free_symbols
set()
>>> PurePoly(x**2 + y).free_symbols
set()
>>> PurePoly(x**2 + y, x).free_symbols
{y}
```

class `sympy.polys.polytools.GroebnerBasis`
Represents a reduced Groebner basis.

contains(poly)
Check if poly belongs the ideal generated by self.

Examples

```
>>> from sympy import groebner
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**3 + y**3 + 3*y
>>> G = groebner([x**2 + y**2 - 1, x*y - 2])
```

```
>>> G.contains(f)
True
>>> G.contains(f + 1)
False
```

fglm(order)
Convert a Groebner basis from one ordering to another.

The FGLM algorithm converts reduced Groebner bases of zero-dimensional ideals from one ordering to another. This method is often used when it is infeasible to compute a Groebner basis with respect to a particular ordering directly.

References

J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import groebner
```

```
>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, x, y, order='grlex')
```

```
>>> list(G.fglm('lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
>>> list(groebner(F, x, y, order='lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
```

is_zero_dimensional

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of F is bounded.

References

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

reduce(expr, auto=True)

Reduces a polynomial modulo a Groebner basis.

Given a polynomial f and a set of polynomials $G = (g_1, \dots, g_n)$, computes a set of quotients $q = (q_1, \dots, q_n)$ and the remainder r such that $f = q_1*f_1 + \dots + q_n*f_n + r$, where r vanishes or r is a completely reduced polynomial with respect to G .

Examples

```
>>> from sympy import groebner, expand
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**4 - x**2 + y**3 + y**2
>>> G = groebner([x**3 - x, y**3 - y])
```

```
>>> G.reduce(f)
([2*x, 1], x**2 + y**2 + y)
>>> Q, r = _
```

```
>>> expand(sum(q*g for q, g in zip(Q, G)) + r)
2*x**4 - x**2 + y**3 + y**2
>>> _ == f
True
```

Extra polynomial manipulation functions

`sympy.polys.polyfuncs.symmetrize(F, *gens, **args)`

Rewrite a polynomial in terms of elementary symmetric polynomials.

A symmetric polynomial is a multivariate polynomial that remains invariant under any variable permutation, i.e., if $f = f(x_1, x_2, \dots, x_n)$, then $f = f(x_{\{i_1\}}, x_{\{i_2\}}, \dots, x_{\{i_n\}})$, where $(i_1, i_2, \dots, i_n)$ is a permutation of $(1, 2, \dots, n)$ (an element of the group S_n).

Returns a tuple of symmetric polynomials $(f_1, f_2, \dots, f_n)$ such that $f = f_1 + f_2 + \dots + f_n$.

Examples

```
>>> from sympy.polys.polyfuncs import symmetrize
>>> from sympy.abc import x, y
```

```
>>> symmetrize(x**2 + y**2)
(-2*x*y + (x + y)**2, 0)
```

```
>>> symmetrize(x**2 + y**2, formal=True)
(s1**2 - 2*s2, 0, [(s1, x + y), (s2, x*y)])
```

```
>>> symmetrize(x**2 - y**2)
(-2*x*y + (x + y)**2, -2*y**2)
```

```
>>> symmetrize(x**2 - y**2, formal=True)
(s1**2 - 2*s2, -2*y**2, [(s1, x + y), (s2, x*y)])
```

`sympy.polys.polyfuncs.horner(f, *gens, **args)`

Rewrite a polynomial in Horner form.

Among other applications, evaluation of a polynomial at a point is optimal when it is applied using the Horner scheme ([1]).

References

[1] - http://en.wikipedia.org/wiki/Horner_scheme

Examples

```
>>> from sympy.polys.polyfuncs import horner
>>> from sympy.abc import x, y, a, b, c, d, e
```

```
>>> horner(9*x**4 + 8*x**3 + 7*x**2 + 6*x + 5)
x*(x*(x*(9*x + 8) + 7) + 6) + 5
```

```
>>> horner(a*x**4 + b*x**3 + c*x**2 + d*x + e)
e + x*(d + x*(c + x*(a*x + b)))
```

```
>>> f = 4*x**2*y**2 + 2*x**2*y + 2*x*y**2 + x*y
```

```
>>> horner(f, wrt=x)
x*(x*y*(4*y + 2) + y*(2*y + 1))
```

```
>>> horner(f, wrt=y)
y*(x*y*(4*x + 2) + x*(2*x + 1))
```

`sympy.polys.polyfuncs.interpolate(data, x)`
Construct an interpolating polynomial for the data points.

Examples

```
>>> from sympy.polys.polyfuncs import interpolate
>>> from sympy.abc import x
```

A list is interpreted as though it were paired with a range starting from 1:

```
>>> interpolate([1, 4, 9, 16], x)
x**2
```

This can be made explicit by giving a list of coordinates:

```
>>> interpolate([(1, 1), (2, 4), (3, 9)], x)
x**2
```

The (x, y) coordinates can also be given as keys and values of a dictionary (and the points need not be equispaced):

```
>>> interpolate([(-1, 2), (1, 2), (2, 5)], x)
x**2 + 1
>>> interpolate({-1: 2, 1: 2, 2: 5}, x)
x**2 + 1
```

`sympy.polys.polyfuncs.viete(f, roots=None, *gens, **args)`
Generate Viete's formulas for f.

Examples

```
>>> from sympy.polys.polyfuncs import viete
>>> from sympy import symbols
```

```
>>> x, a, b, c, r1, r2 = symbols('x,a:c,r1:3')
```

```
>>> viete(a*x**2 + b*x + c, [r1, r2], x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

Domain constructors

`sympy.polys.constructor.construct_domain(obj, **args)`
Construct a minimal domain for the list of coefficients.

Algebraic number fields

`sympy.polys.numberfields.minimal_polynomial(ex, x=None, compose=True, polys=False, domain=None)`

Computes the minimal polynomial of an algebraic element.

Parameters `ex` : Expr

Element or expression whose minimal polynomial is to be calculated.

`x` : Symbol, optional

Independent variable of the minimal polynomial

compose : boolean, optional (default=True)

Method to use for computing minimal polynomial. If `compose=True` (default) then `_minpoly_compose` is used, if `compose=False` then groebner bases are used.

polys : boolean, optional (default=False)

If True returns a Poly object else an Expr object.

domain : Domain, optional

Ground domain

Notes

By default `compose=True`, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If `compose=False`, a bottom-up algorithm is used with groebner. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y
```

```
>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`sympy.polys.numberfields.minpoly(ex, x=None, compose=True, polys=False, domain=None)`

Computes the minimal polynomial of an algebraic element.

Parameters `ex` : Expr

Element or expression whose minimal polynomial is to be calculated.

x : Symbol, optional

Independent variable of the minimal polynomial

compose : boolean, optional (default=True)

Method to use for computing minimal polynomial. If compose=True (default) then `_minpoly_compose` is used, if compose=False then groebner bases are used.

polys : boolean, optional (default=False)

If True returns a Poly object else an Expr object.

domain : Domain, optional

Ground domain

Notes

By default compose=True, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If compose=False, a bottom-up algorithm is used with groebner. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y
```

```
>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`sympy.polys.numberfields.primitive_element(extension, x=None, **args)`
Construct a common number field for all extensions.

`sympy.polys.numberfields.field_isomorphism(a, b, **args)`
Construct an isomorphism between two number fields.

`sympy.polys.numberfields.to_number_field(extension, theta=None, **args)`
Express *extension* in the field generated by *theta*.

`sympy.polys.numberfields.isolate(alg, eps=None, fast=False)`
Give a rational isolating interval for an algebraic number.

class `sympy.polys.numberfields.AlgebraicNumber`
Class for representing algebraic numbers in SymPy.

as_expr(x=None)
Create a Basic expression from self.

as_poly(x=None)
Create a Poly instance from self.

coeffs()
Returns all SymPy coefficients of an algebraic number.

is_aliased
Returns True if alias was set.

native_coeffs()
Returns all native coefficients of an algebraic number.

to_algebraic_integer()
Convert self to an algebraic integer.

Monomials encoded as tuples

class `sympy.polys.monomials.Monomial`(monom, gens=None)
Class representing a monomial, i.e. a product of powers.

`sympy.polys.monomials.itermonomials`(variables, degree)
Generate a set of monomials of the given total degree or less.

Given a set of variables V and a total degree N generate a set of monomials of degree at most N . The total number of monomials in commutative variables is huge and is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

For example if we would like to generate a dense polynomial of a total degree $N = 50$ in 5 variables, assuming that exponents and all of coefficients are 32-bit long and stored in an array we would need almost 80 GiB of memory! Fortunately most polynomials, that we will encounter, are sparse.

Examples

Consider monomials in commutative variables x and y and non-commutative variables a and b :

```
>>> from sympy import symbols
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> sorted(itermonomials([x, y], 2), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]

>>> sorted(itermonomials([x, y], 3), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2, x**3, x**2*y, x*y**2, y**3]

>>> a, b = symbols('a, b', commutative=False)
>>> itermonomials([a, b, x], 2)
{1, a, a**2, b, b**2, x, x**2, a*b, b*a, x*a, x*b}
```

`sympy.polys.monomials.monomial_count`(V, N)
Computes the number of monomials.

The number of monomials is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

where N is a total degree and V is a set of variables.

Examples

```
>>> from sympy.polys.monomials import itermonomials, monomial_count
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y
```

```
>>> monomial_count(2, 2)
6
```

```
>>> M = itermonomials([x, y], 2)
```

```
>>> sorted(M, key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
>>> len(M)
6
```

Orderings of monomials

class sympy.polys.orderings.LexOrder
Lexicographic order of monomials.

class sympy.polys.orderings.GradedLexOrder
Graded lexicographic order of monomials.

class sympy.polys.orderings.ReversedGradedLexOrder
Reversed graded lexicographic order of monomials.

Formal manipulation of roots of polynomials

sympy.polys.rootoftools.**rootof**(f, x, index=None, radicals=True, expand=True)
An indexed root of a univariate polynomial.

Returns either a ComplexRootOf object or an explicit expression involving radicals.

Parameters **f** : Expr

Univariate polynomial.

x : Symbol, optional

Generator for f.

index : int or Integer

radicals : bool

Return a radical expression if possible.

expand : bool

Expand f.

class sympy.polys.rootoftools.**RootOf**

Represents a root of a univariate polynomial.

Base class for roots of different kinds of polynomials. Only complex roots are currently supported.

class sympy.polys.rootoftools.**ComplexRootOf**

Represents an indexed complex root of a polynomial.

Roots of a univariate polynomial separated into disjoint real or complex intervals and indexed in a fixed order. Currently only rational coefficients are allowed. Can be imported as CRootOf.

See also:

eval_approx, eval_rational, _eval_evalf

Examples

```
>>> from sympy import CRootOf, rootof
>>> from sympy.abc import x
```

CRootOf is a way to reference a particular root of a polynomial. If there is a rational root, it will be returned:

```
>>> CRootOf(x**2 - 4, 0)
-2
```

Whether roots involving radicals are returned or not depends on whether the radicals flag is true (which is set to True with rootof):

```
>>> CRootOf(x**2 - 3, 0)
CRootOf(x**2 - 3, 0)
>>> CRootOf(x**2 - 3, 0, radicals=True)
-sqrt(3)
>>> rootof(x**2 - 3, 0)
-sqrt(3)
```

The following cannot be expressed in terms of radicals:

```
>>> r = rootof(4*x**5 + 16*x**3 + 12*x**2 + 7, 0); r
CRootOf(4*x**5 + 16*x**3 + 12*x**2 + 7, 0)
```

The root bounds can be seen, however, and they are used by the evaluation methods to get numerical approximations for the root.

```
>>> interval = r._get_interval(); interval
(-1, 0)
>>> r.evalf(2)
-0.98
```

The evalf method refines the width of the root bounds until it guarantees that any decimal approximation within those bounds will satisfy the desired precision. It then stores the refined interval so subsequent requests at or below the requested precision will not have to recompute the root bounds and will return very quickly.

Before evaluation above, the interval was

```
>>> interval
(-1, 0)
```

After evaluation it is now

```
>>. r._get_interval() (-165/169, -206/211)
```

To reset all intervals for a given polynomial, the `reset` method can be called from any `CRootOf` instance of the polynomial:

```
>>> r._reset()
>>> r._get_interval()
(-1, 0)
```

The `evalapprox` method will also find the root to a given precision but the interval is not modified unless the search for the root fails to converge within the root bounds. And the secant method is used to find the root. (The `evalf` method uses bisection and will always update the interval.)

```
>>> r.eval_approx(2)
-0.98
```

The interval needed to be slightly updated to find that root:

```
>>> r._get_interval()
(-1, -1/2)
```

The `evalfrational` will compute a rational approximation of the root to the desired accuracy or precision.

```
>>> r.eval_rational(n=2)
-69629/71318
```

```
>>> t = CRootOf(x**3 + 10*x + 1, 1)
>>> t.eval_rational(1e-1)
15/256 - 805*I/256
>>> t.eval_rational(1e-1, 1e-4)
3275/65536 - 414645*I/131072
>>> t.eval_rational(1e-4, 1e-4)
6545/131072 - 414645*I/131072
>>> t.eval_rational(n=2)
104755/2097152 - 6634255*I/2097152
```

class `sympy.polys.rootoftools.RootSum`

Represents a sum of all roots of a univariate polynomial.

Symbolic root-finding algorithms

`sympy.polys.polyroots.roots(f, *gens, **flags)`

Computes symbolic roots of a univariate polynomial.

Given a univariate polynomial f with symbolic coefficients (or a list of the polynomial's coefficients), returns a dictionary with its roots and their multiplicities.

Only roots expressible via radicals will be returned. To get a complete set of roots use `RootOf` class or numerical methods instead. By default cubic and quartic formulas are used in the algorithm. To disable them because of unreadable output set `cubics=False`

or `quartics=False` respectively. If cubic roots are real but are expressed in terms of complex numbers (casus irreducibilis [1]) the `trig` flag can be set to `True` to have the solutions returned in terms of cosine and inverse cosine functions.

To get roots from a specific domain set the `filter` flag with one of the following specifiers: `Z`, `Q`, `R`, `I`, `C`. By default all roots are returned (this is equivalent to setting `filter='C'`).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the `multiple` flag to `True`; the list will have identical roots appearing next to each other in the result. (For a given `Poly`, the `all_roots` method will give the roots in sorted numerical order.)

References

1. http://en.wikipedia.org/wiki/Cubic_function#Trigonometric_.28and_hyperbolic.29_method

Examples

```
>>> from sympy import Poly, roots
>>> from sympy.abc import x, y
```

```
>>> roots(x**2 - 1, x)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-1, x)
>>> roots(p)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-y, x, y)
```

```
>>> roots(Poly(p, x))
{-sqrt(y): 1, sqrt(y): 1}
```

```
>>> roots(x**2 - y, x)
{-sqrt(y): 1, sqrt(y): 1}
```

```
>>> roots([1, 0, -1])
{-1: 1, 1: 1}
```

Special polynomials

`sympy.polys.specialpolys.swinnerton_dyer_poly(n, x=None, polys=False)`
Generates n -th Swinnerton-Dyer polynomial in x .

Parameters `n` : int

n decides the order of polynomial

`x` : optional

`polys` : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.specialpolys.interpolating_poly(n, x, X='x', Y='y')`
Construct Lagrange interpolating polynomial for n data points.

`sympy.polys.specialpolys.cyclotomic_poly(n, x=None, polys=False)`
Generates cyclotomic polynomial of order n in x .

Parameters `n` : int

n decides the order of polynomial

`x` : optional

`polys` : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.specialpolys.symmetric_poly(n, *gens, **args)`
Generates symmetric polynomial of order n .

Returns a Poly object when `polys=True`, otherwise (default) returns an expression.

`sympy.polys.specialpolys.random_poly(x, n, inf, sup, domain=ZZ, polys=False)`
Generates a polynomial of degree n with coefficients in $[inf, sup]$.

Parameters `x`

x is the independent term of polynomial

`n` : int

n decides the order of polynomial

`inf`

Lower limit of range in which coefficients lie

`sup`

Upper limit of range in which coefficients lie

`domain` : optional

Decides what ring the coefficients are supposed to belong. Default is set to Integers.

`polys` : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

Orthogonal polynomials

`sympy.polys.orthopolys.chebyshevt_poly(n, x=None, polys=False)`
Generates Chebyshev polynomial of the first kind of degree n in x .

Parameters `n` : int

n decides the degree of polynomial

`x` : optional

`polys` : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.chebyshev_poly(n, x=None, polys=False)`
Generates Chebyshev polynomial of the second kind of degree n in x .

Parameters n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.gegenbauer_poly(n, a, x=None, polys=False)`
Generates Gegenbauer polynomial of degree n in x .

Parameters n : int

n decides the degree of polynomial

x : optional

a

Decides minimal domain for the list of coefficients.

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.hermite_poly(n, x=None, polys=False)`
Generates Hermite polynomial of degree n in x .

Parameters n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.jacobi_poly(n, a, b, x=None, polys=False)`
Generates Jacobi polynomial of degree n in x .

Parameters n : int

n decides the degree of polynomial

a

Lower limit of minimal domain for the list of coefficients.

b

Upper limit of minimal domain for the list of coefficients.

x : optional

polys : bool, optional

`polys=True` returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.legendre_poly(n, x=None, polys=False)`
Generates Legendre polynomial of degree n in x .

Parameters n : int

n decides the degree of polynomial

x : optional

polys : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

`sympy.polys.orthopolys.laguerre_poly(n, x=None, alpha=None, polys=False)`

Generates Laguerre polynomial of degree n in x .

Parameters **n** : int

n decides the degree of polynomial

x : optional

alpha

Decides minimal domain for the list of coefficients.

polys : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

Manipulation of rational functions

`sympy.polys.rationaltools.together(expr, deep=False)`

Denest and combine rational expressions using symbolic methods.

This function takes an expression or a container of expressions and puts it (them) together by denesting and combining rational subexpressions. No heroic measures are taken to minimize degree of the resulting numerator and denominator. To obtain completely reduced expression use `cancel()`. However, `together()` (page 875) can preserve as much as possible of the structure of the input expression in the output (no expansion is performed).

A wide variety of objects can be put together including lists, tuples, sets, relational objects, integrals and others. It is also possible to transform interior of function applications, by setting deep flag to True.

By definition, `together()` (page 875) is a complement to `apart()`, so `apart(together(expr))` should return `expr` unchanged. Note however, that `together()` (page 875) uses only symbolic methods, so it might be necessary to use `cancel()` to perform algebraic simplification and minimize degree of the numerator and denominator.

Examples

```
>>> from sympy import together, exp
>>> from sympy.abc import x, y, z
```

```
>>> together(1/x + 1/y)
(x + y)/(x*y)
>>> together(1/x + 1/y + 1/z)
(x*y + x*z + y*z)/(x*y*z)
```

```
>>> together(1/(x*y) + 1/y**2)
(x + y)/(x*y**2)
```

```
>>> together(1/(1 + 1/x) + 1/(1 + 1/y))
(x*(y + 1) + y*(x + 1))/((x + 1)*(y + 1))
```

```
>>> together(exp(1/x + 1/y))
exp(1/y + 1/x)
>>> together(exp(1/x + 1/y), deep=True)
exp((x + y)/(x*y))
```

```
>>> together(1/exp(x) + 1/(x*exp(x)))
(x + 1)*exp(-x)/x
```

```
>>> together(1/exp(2*x) + 1/(x*exp(3*x)))
(x*exp(x) + 1)*exp(-3*x)/x
```

Partial fraction decomposition

`sympy.polys.partfrac.apart(f, x=None, full=False, **options)`
Compute partial fraction decomposition of a rational function.

Given a rational function f , computes the partial fraction decomposition of f . Two algorithms are available: One is based on the undertermined coefficients method, the other is Bronstein's full partial fraction decomposition algorithm.

The undetermined coefficients method (selected by `full=False`) uses polynomial factorization (and therefore accepts the same options as `factor`) for the denominator. Per default it works over the rational numbers, therefore decomposition of denominators with non-rational roots (e.g. irrational, complex roots) is not supported by default (see options of `factor`).

Bronstein's algorithm can be selected by using `full=True` and allows a decomposition of denominators with non-rational roots. A human-readable result can be obtained via `doit()` (see examples below).

See also:

[apart_list](#) (page 877), [assemble_partfrac_list](#) (page 878)

Examples

```
>>> from sympy.polys.partfrac import apart
>>> from sympy.abc import x, y
```

By default, using the undetermined coefficients method:

```
>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)
```

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*_a*y/3 - y/3)/(-_a + x)))
```

Calling `doit()` yields a human-readable result:

```
>>> apart(y/(x**2 + x + 1), x, full=True).doit()
(-y/3 - 2*y*(-1/2 - sqrt(3)*I/2)/3)/(x + 1/2 + sqrt(3)*I/2) + (-y/3 -
 2*y*(-1/2 + sqrt(3)*I/2)/3)/(x + 1/2 - sqrt(3)*I/2)
```

`sympy.polys.partfrac.apart_list(f, x=None, dummies=None, **options)`

Compute partial fraction decomposition of a rational function and return the result in structured form.

Given a rational function `f` compute the partial fraction decomposition of `f`. Only Bronstein's full partial fraction decomposition algorithm is supported by this method. The return value is highly structured and perfectly suited for further algorithmic treatment rather than being human-readable. The function returns a tuple holding three elements:

- The first item is the common coefficient, free of the variable x used for decomposition. (It is an element of the base field K .)
- The second item is the polynomial part of the decomposition. This can be the zero polynomial. (It is an element of $K[x]$.)
- The third part itself is a list of quadruples. Each quadruple has the following elements in this order:
 - The (not necessarily irreducible) polynomial D whose roots w_i appear in the linear denominator of a bunch of related fraction terms. (This item can also be a list of explicit roots. However, at the moment `apart_list` never returns a result this way, but the related `assemble_partfrac_list` function accepts this format as input.)
 - The numerator of the fraction, written as a function of the root w
 - The linear denominator of the fraction excluding its power exponent, written as a function of the root w .
 - The power to which the denominator has to be raised.

One can always rebuild a plain expression by using the function `assemble_partfrac_list`.

See also:

`apart` (page 876), `assemble_partfrac_list` (page 878)

References

1. [Bronstein93] (page 1866)

[Bronstein93] (page 1866)

Examples

A first example:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, t
```

```
>>> f = (2*x**3 - 2*x) / (x**2 - 2*x + 1)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(2*x + 4, x, domain='ZZ'),
[(Poly(_w - 1, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
2*x + 4 + 4/(x - 1)
```

Second example:

```
>>> f = (-2*x - 2*x**2) / (3*x**2 - 6*x)
>>> pfd = apart_list(f)
>>> pfd
(-1,
Poly(2/3, x, domain='QQ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 2), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-2/3 - 2/(x - 2)
```

Another example, showing symbolic parameters:

```
>>> pfd = apart_list(t/(x**2 + x + t), x)
>>> pfd
(1,
Poly(0, x, domain='ZZ[t]'),
[(Poly(_w**2 + _w + t, _w, domain='ZZ[t]'),
Lambda(_a, -2*_a*t/(4*t - 1) - t/(4*t - 1)),
Lambda(_a, -_a + x),
1)])
```

```
>>> assemble_partfrac_list(pfd)
RootSum(_w**2 + _w + t, Lambda(_a, (-2*_a*t/(4*t - 1) - t/(4*t - 1))/(-_a + x)))
```

This example is taken from Bronstein's original paper:

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
(Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a + x), 2),
(Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

`sympy.polys.partfrac.assemble_partfrac_list(partial_list)`

Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.

See also:

`apart` (page 876), `apart_list` (page 877)

Examples

This example is taken from Bronstein's original paper:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, y
```

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
 Poly(0, x, domain='ZZ'),
 [(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
 (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a + x), 2),
 (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

If we happen to know some roots we can provide them easily inside the structure:

```
>>> pfd = apart_list(2/(x**2-2))
>>> pfd
(1,
 Poly(0, x, domain='ZZ'),
 [(Poly(_w**2 - 2, _w, domain='ZZ'),
 Lambda(_a, _a/2),
 Lambda(_a, -_a + x),
 1)])
```

```
>>> pfda = assemble_partfrac_list(pfd)
>>> pfda
RootSum(_w**2 - 2, Lambda(_a, _a/(-_a + x)))/2
```

```
>>> pfda.doit()
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

```
>>> from sympy import Dummy, Poly, Lambda, sqrt
>>> a = Dummy("a")
>>> pfd = (1, Poly(0, x, domain='ZZ'), [(sqrt(2), -sqrt(2)), Lambda(a, a/2),
↳ Lambda(a, -a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

Dispersion of Polynomials

`sympy.polys.dispersion.dispersionset(p, q=None, *gens, **args)`

Compute the dispersion set of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

See also:

[dispersion](#) (page 808)

References

1. [\[ManWright94\]](#) (page 1867)
2. [\[Koepf98\]](#) (page 1867)
3. [\[Abramov71\]](#) (page 1867)
4. [\[Man93\]](#) (page 1867)

[\[ManWright94\]](#) (page 1867), [\[Koepf98\]](#) (page 1867), [\[Abramov71\]](#) (page 1867), [\[Man93\]](#) (page 1867)

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`sympy.polys.dispersion.dispersion(p, q=None, *gens, **args)`

Compute the dispersion of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\text{dis}(f, g)$ is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial $\text{dis}(f) := \text{dis}(f, f)$. Note that we make the definition $\max\{\} := -\infty$.

See also:

[dispersionset](#) (page 809)

References

1. [\[ManWright94\]](#) (page 1867)
2. [\[Koepf98\]](#) (page 1867)
3. [\[Abramov71\]](#) (page 1867)
4. [\[Man93\]](#) (page 1867)

[\[ManWright94\]](#) (page 1867), [\[Koepf98\]](#) (page 1867), [\[Abramov71\]](#) (page 1867), [\[Man93\]](#) (page 1867)

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be $-\infty$ as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

AGCA - Algebraic Geometry and Commutative Algebra Module

Introduction

Algebraic geometry is a mixture of the ideas of two Mediterranean cultures. It is the superposition of the Arab science of the lightening calculation of the solutions of equations over the Greek art of position and shape. This tapestry was originally woven on European soil and is still being refined under the influence of international fashion. Algebraic geometry studies the delicate balance between the geometrically plausible and the algebraically possible. Whenever one side of this mathematical teeter-totter outweighs the other, one immediately loses interest and runs off in search of a more exciting amusement.

George R. Kempf 1944 - 2002

Algebraic Geometry refers to the study of geometric problems via algebraic methods (and sometimes vice versa). While this is a rather old topic, algebraic geometry as understood today is very much a 20th century development. Building on ideas of e.g. Riemann and Dedekind, it was realized that there is an intimate connection between properties of the set

of solutions of a system of polynomial equations (called an algebraic variety) and the behavior of the set of polynomial functions on that variety (called the coordinate ring).

As in many geometric disciplines, we can distinguish between local and global questions (and methods). Local investigations in algebraic geometry are essentially equivalent to the study of certain rings, their ideals and modules. This latter topic is also called commutative algebra. It is the basic local toolset of algebraic geometers, in much the same way that differential analysis is the local toolset of differential geometers.

A good conceptual introduction to commutative algebra is [Atiyah69] (page 1867). An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008] (page 1866).

This module aims to eventually allow expression and solution of both local and global geometric problems, both in the classical case over a field and in the more modern arithmetic cases. So far, however, there is no geometric functionality at all. Currently the module only provides tools for computational commutative algebra over fields.

All code examples assume:

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(use_unicode=True, wrap_line=False, no_global=True)
```

Reference

In this section we document the usage of the AGCA module. For convenience of the reader, some definitions and examples/explanations are interspersed.

Base Rings

Almost all computations in commutative algebra are relative to a “base ring”. (For example, when asking questions about an ideal, the base ring is the ring the ideal is a subset of.) In principle all polys “domains” can be used as base rings. However, useful functionality is only implemented for polynomial rings over fields, and various localizations and quotients thereof.

As demonstrated in the examples below, the most convenient method to create objects you are interested in is to build them up from the ground field, and then use the various methods to create new objects from old. For example, in order to create the local ring of the nodal cubic $y^2 = x^3$ at the origin, over $\mathbb{Q}$, you do:

```
>>> lr = QQ.old_poly_ring(x, y, order="ilex") / [y**2 - x**3]
>>> lr
Q[x, y, order=ilex]
```

$$\left\langle \begin{array}{cc} 3 & 2 \\ -x & +y \end{array} \right\rangle$$

Note how the python list notation can be used as a short cut to express ideals. You can use the `convert` method to return ordinary sympy objects into objects understood by the AGCA module (although in many cases this will be done automatically – for example the list was automatically turned into an ideal, and in the process the symbols x and y were automatically converted into other representations). For example:

```
>>> X, Y = lr.convert(x), lr.convert(y) ; X
x +  $\begin{pmatrix} 3 & 2 \\ -x & +y \end{pmatrix}$ 
>>> x**3 == y**2
False
>>> X**3 == Y**2
True
```

When no localisation is needed, a more mathematical notation can be used. For example, let us create the coordinate ring of three-dimensional affine space $\mathbb{A}^3$:

```
>>> ar = QQ.old_poly_ring(x, y, z); ar
Q[x, y, z]
```

For more details, refer to the following class documentation. Note that the base rings, being domains, are the main point of overlap between the AGCA module and the rest of the polys module. All domains are documented in detail in the polys reference, so we show here only an abridged version, with the methods most pertinent to the AGCA module.

class sympy.polys.domains.ring.Ring

Represents a ring domain.

Attributes

alias	
dtype	
one	
rep	
zero	

free_module(rank)

Generate a free module of rank rank over self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

ideal(*gens)

Generate an ideal of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

quotient_ring(e)

Form a quotient ring of self.

Here e can be an ideal or an iterable.

```
>>> from sympy.abc import x
>>> from sympy import QQ
```

(continues on next page)

(continued from previous page)

```
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

`sympy.polys.domains.polynomialring.PolynomialRing(domain_or_ring, symbols=None, order=None)`

A class for representing multivariate polynomial rings.

Attributes

alias	
domain	
dtype	
gens	
ngens	
rep	
symbols	

class `sympy.polys.domains.quotientring.QuotientRing(ring, ideal)`

Class representing (commutative) quotient rings.

You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>
```

```
>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>
```

Attributes:

- `ring` - the base ring
- `base_ideal` - the ideal used to form the quotient

Attributes

alias	
one	
rep	
zero	

Modules, Ideals and their Elementary Properties

Let A be a ring. An A -module is a set M , together with two binary operations $+: M \times M \rightarrow M$ and $\times: R \times M \rightarrow M$ called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69] (page 1867). In this way modules are a direct generalisation of both vector spaces (A being a field) and abelian groups ($A = \mathbb{Z}$). A submodule of the A -module M is a subset $N \subset M$, such that the binary operations restrict to N , and N becomes an A -module with these operations.

The ring A itself has a natural A -module structure where addition and multiplication in the module coincide with addition and multiplication in the ring. This A -module is also written as A . An A -submodule of A is called an ideal of A . Ideals come up very naturally in algebraic geometry. More general modules can be seen as a technically convenient “elbow room” beyond talking only about ideals.

If M, N are A -modules, then there is a natural (componentwise) A -module structure on $M \times N$. Similarly there are A -module structures on cartesian products of more components. (For the categorically inclined: the cartesian product of finitely many A -modules, with this A -module structure, is the finite biproduct in the category of all A -modules. With infinitely many components, it is the direct product (but the infinite direct sum has to be constructed differently).) As usual, repeated product of the A -module M is denoted $M, M^2, M^3 \dots$, or M^I for arbitrary index sets I .

An A -module M is called free if it is isomorphic to the A -module A^I for some (not necessarily finite) index set I (refer to the next section for a definition of isomorphism). The cardinality of I is called the rank of M ; one may prove this is well-defined. In general, the AGCA module only works with free modules of finite rank, and other closely related modules. The easiest way to create modules is to use member methods of the objects they are made up from. For example, let us create a free module of rank 4 over the coordinate ring of $\mathbb{A}^2$ we created above, together with a submodule:

```
>>> F = ar.free_module(4) ; F
      4
Q[x, y, z]

>>> S = F.submodule([1, x, x**2, x**3], [0, 1, 0, y]) ; S
<[
  2   3
  1, x, x , x ], [0, 1, 0, y]>
```

Note how python lists can be used as a short-cut notation for module elements (vectors). As usual, the `convert` method can be used to convert sympy/python objects into the internal AGCA representation (see detailed reference below).

Here is the detailed documentation of the classes for modules, free modules, and submodules:

```
class sympy.polys.agca.modules.Module(ring)
    Abstract base class for modules.
```

Do not instantiate - use ring explicit constructors instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

Attributes:

- `dtype` - type of elements
- `ring` - containing ring

Non-implemented methods:

- `submodule`
- `quotient_module`
- `is_zero`
- `is_submodule`
- `multiply_ideal`

The method `convert` likely needs to be changed in subclasses.

`contains(elem)`

Return True if `elem` is an element of this module.

`convert(elem, M=None)`

Convert `elem` into internal representation of this module.

If `M` is not None, it should be a module containing it.

`identity_hom()`

Return the identity homomorphism on `self`.

`is_submodule(other)`

Returns True if `other` is a submodule of `self`.

`is_zero()`

Returns True if `self` is a zero module.

`multiply_ideal(other)`

Multiply `self` by the ideal `other`.

`quotient_module(other)`

Generate a quotient module.

`submodule(*gens)`

Generate a submodule.

`subset(other)`

Returns True if `other` is a subset of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.subset([(1, x), (x, 2)])
True
>>> F.subset([(1/x, x), (x, 2)])
False
```

`class sympy.polys.agca.modules.FreeModule(ring, rank)`

Abstract base class for free modules.

Additional attributes:

- rank - rank of the free module

Non-implemented methods:

- submodule

basis()

Return a set of basis elements.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(3).basis()
([1, 0, 0], [0, 1, 0], [0, 0, 1])
```

convert(elem, M=None)

Convert elem into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.convert([1, 0])
[1, 0]
```

dtype

alias of FreeModuleElement

identity_hom()

Return the identity homomorphism on self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).identity_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

is_submodule(other)

Returns True if other is a submodule of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> F.is_submodule(F)
True
>>> F.is_submodule(M)
True
>>> M.is_submodule(F)
False
```

is_zero()

Returns True if self is a zero module.

(If, as this implementation assumes, the coefficient ring is not the zero ring, then this is equivalent to the rank being zero.)

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(0).is_zero()
True
>>> QQ.old_poly_ring(x).free_module(1).is_zero()
False
```

multiply_ideal(other)
Multiply self by the ideal other.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x)
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.multiply_ideal(I)
<[x, 0], [0, x]>
```

quotient_module(submodule)
Return a quotient module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2)
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more concisely, using the overloaded division operator:

```
>>> QQ.old_poly_ring(x).free_module(2) / [[1, x], [x, 2]]
QQ[x]**2/<[1, x], [x, 2]>
```

class sympy.polys.agca.modules.**SubModule**(gens, container)
Base class for submodules.

Attributes:

- container - containing module
- gens - generators (subset of containing module)
- rank - rank of containing module

Non-implemented methods:

- _contains
- _syzygies
- _in_terms_of_generators
- _intersect
- _module_quotient

Methods that likely need change in subclasses:

- reduce_element

convert(elem, M=None)
Convert elem into the internal representation.
Mostly called implicitly.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, x])
>>> M.convert([2, 2*x])
[2, 2*x]
```

identity_hom()

Return the identity homomorphism on self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).identity_hom()
Matrix([
[1, 0], : <[x, x]> -> <[x, x]>
[0, 1]])
```

in_terms_of_generators(e)

Express element e of self in terms of the generators.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([1, 0], [1, 1])
>>> M.in_terms_of_generators([x, x**2])
[-x**2 + x, x**2]
```

inclusion_hom()

Return a homomorphism representing the inclusion map of self.

That is, the natural map from self to self.container.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).inclusion_hom()
Matrix([
[1, 0], : <[x, x]> -> QQ[x]**2
[0, 1]])
```

intersect(other, **options)

Returns the intersection of self with submodule other.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> F.submodule([x, x]).intersect(F.submodule([y, y]))
<[x*y, x*y]>
```

Some implementation allow further options to be passed. Currently, to only one implemented is `relations=True`, in which case the function will return a triple (`res`, `rela`, `relb`), where `res` is the intersection module, and `rela` and `relb` are lists of coefficient vectors, expressing the generators of `res` in terms of the generators of `self` (`rela`) and `other` (`relb`).

```
>>> F.submodule([x, x]).intersect(F.submodule([y, y]), relations=True)
(<[x*y, x*y]>, [(y,)], [(x,)])
```

The above result says: the intersection module is generated by the single element $(-xy, -xy) = -y(x, x) = -x(y, y)$, where (x, x) and (y, y) respectively are the unique

generators of the two modules being intersected.

is_full_module()

Return True if self is the entire free module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

is_submodule(other)

Returns True if other is a submodule of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> N = M.submodule([2*x, x**2])
>>> M.is_submodule(M)
True
>>> M.is_submodule(N)
True
>>> N.is_submodule(M)
False
```

is_zero()

Return True if self is a zero module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_zero()
False
>>> F.submodule([0, 0]).is_zero()
True
```

module_quotient(other, **options)

Returns the module quotient of self by submodule other.

That is, if self is the module M and other is N , then return the ideal $\{f \in R \mid fN \subset M\}$.

```
>>> from sympy import QQ
>>> from sympy.abc import x, y
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> S = F.submodule([x*y, x*y])
>>> T = F.submodule([x, x])
>>> S.module_quotient(T)
<y>
```

Some implementations allow further options to be passed. Currently, the only one implemented is `relations=True`, which may only be passed if other is principal. In this case the function will return a pair (res, rel) where res is the ideal, and rel is a list of coefficient vectors, expressing the generators of the ideal, multiplied by the generator of other in terms of generators of self.

```
>>> S.module_quotient(T, relations=True)
(<y>, [[1]])
```

This means that the quotient ideal is generated by the single element y , and that $y(x, x) = 1(xy, xy)$, (x, x) and (xy, xy) being the generators of T and S , respectively.

multiply_ideal(I)

Multiply self by the ideal I.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**2)
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, 1])
>>> I*M
<[x**2, x**2]>
```

quotient_module(other, **opts)

Return a quotient module.

This is the same as taking a submodule of a quotient of the containing module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S1 = F.submodule([x, 1])
>>> S2 = F.submodule([x**2, x])
>>> S1.quotient_module(S2)
<[x, 1] + <[x**2, x]>>
```

Or more coincisely, using the overloaded division operator:

```
>>> F.submodule([x, 1]) / [(x**2, x)]
<[x, 1] + <[x**2, x]>>
```

reduce_element(x)

Reduce the element x of our ring modulo the ideal self.

Here “reduce” has no specific meaning, it could return a unique normal form, simplify the expression a bit, or just do nothing.

submodule(*gens)

Generate a submodule.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([x, 1])
>>> M.submodule([x**2, x])
<[x**2, x]>
```

syzygy_module(opts)**

Compute the syzygy module of the generators of self.

Suppose M is generated by $f_1, \dots, f_n$ over the ring R . Consider the homomorphism $\phi : R^n \rightarrow M$, given by sending $(r_1, \dots, r_n) \rightarrow r_1 f_1 + \dots + r_n f_n$. The syzygy module is defined to be the kernel of ϕ .

The syzygy module is zero iff the generators generate freely a free submodule:

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([1, 0], [1, 1]).syzygy_
    module().is_zero()
True
```

A slightly more interesting example:

```
>>> M = QQ.old_poly_ring(x, y).free_module(2).submodule([x, 2*x], [y, 2*y])
>>> S = QQ.old_poly_ring(x, y).free_module(2).submodule([y, -x])
>>> M.syzygy_module() == S
True
```

union(other)

Returns the module generated by the union of self and other.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(1)
>>> M = F.submodule([x**2 + x]) # <x(x+1)>
>>> N = F.submodule([x**2 - 1]) # <(x-1)(x+1)>
>>> M.union(N) == F.submodule([x+1])
True
```

Ideals are created very similarly to modules. For example, let's verify that the nodal cubic is indeed singular at the origin:

```
>>> I = lr.ideal(x, y)
>>> I == lr.ideal(x)
False

>>> I == lr.ideal(y)
False
```

We are using here the fact that a curve is non-singular at a point if and only if the maximal ideal of the local ring is principal, and that in this case at least one of x and y must be generators.

This is the detailed documentation of the class ideal. Please note that most of the methods regarding properties of ideals (primality etc.) are not yet implemented.

class sympy.polys.agca.ideals.**Ideal**(ring)

Abstract base class for ideals.

Do not instantiate - use explicit constructors in the ring class instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).ideal(x+1)
<x + 1>
```

Attributes

- ring - the ring this ideal belongs to

Non-implemented methods:

- _contains_elem
- _contains_ideal
- _quotient

- `_intersect`
- `_union`
- `_product`
- `is_whole_ring`
- `is_zero`
- `is_prime`, `is_maximal`, `is_primary`, `is_radical`
- `is_principal`
- `height`, `depth`
- `radical`

Methods that likely should be overridden in subclasses:

- `reduce_element`

contains(elem)

Return True if elem is an element of this ideal.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x+1, x-1).contains(3)
True
>>> QQ.old_poly_ring(x).ideal(x**2, x**3).contains(x)
False
```

depth()

Compute the depth of self.

height()

Compute the height of self.

intersect(J)

Compute the intersection of self with ideal J.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x).intersect(R.ideal(y))
<x*y>
```

is_maximal()

Return True if self is a maximal ideal.

is_primary()

Return True if self is a primary ideal.

is_prime()

Return True if self is a prime ideal.

is_principal()

Return True if self is a principal ideal.

is_radical()

Return True if self is a radical ideal.

is_whole_ring()

Return True if self is the whole ring.

is_zero()

Return True if self is the zero ideal.

product(J)

Compute the ideal product of self and J.

That is, compute the ideal generated by products xy , for x an element of self and $y \in J$.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).ideal(x).product(QQ.old_poly_ring(x, y).ideal(y))
<x*y>
```

quotient(J, **opts)

Compute the ideal quotient of self by J.

That is, if self is the ideal I , compute the set $I : J = \{x \in R \mid xJ \subset I\}$.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x*y).quotient(R.ideal(x))
<y>
```

radical()

Compute the radical of self.

reduce_element(x)

Reduce the element x of our ring modulo the ideal self.

Here “reduce” has no specific meaning: it could return a unique normal form, simplify the expression a bit, or just do nothing.

saturate(J)

Compute the ideal saturation of self by J.

That is, if self is the ideal I , compute the set $I : J^\infty = \{x \in R \mid xJ^n \subset I \text{ for some } n\}$.

subset(other)

Returns True if other is a subset of self.

Here other may be an ideal.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x+1)
>>> I.subset([x**2 - 1, x**2 + 2*x + 1])
True
>>> I.subset([x**2 + 1, x + 1])
False
>>> I.subset(QQ.old_poly_ring(x).ideal(x**2 - 1))
True
```

union(J)

Compute the ideal generated by the union of self and J.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2 - 1).union(QQ.old_poly_ring(x).
    ideal((x+1)**2)) == QQ.old_poly_ring(x).ideal(x+1)
True
```

If M is an A -module and N is an A -submodule, we can define two elements x and y of M to be equivalent if $x - y \in N$. The set of equivalence classes is written M/N , and has a natural

A -module structure. This is called the quotient module of M by N . If K is a submodule of M containing N , then K/N is in a natural way a submodule of M/N . Such a module is called a subquotient. Here is the documentation of quotient and subquotient modules:

class sympy.polys.agca.modules.QuotientModule(ring, base, submodule)

Class for quotient modules.

Do not instantiate this directly. For subquotients, see the SubQuotientModule class.

Attributes:

- base - the base module we are a quotient of
- killed_module - the submodule used to form the quotient
- rank of the base

convert(elem, M=None)

Convert elem into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> F.convert([1, 0])
[1, 0] + <[1, 2], [1, x]>
```

dtype

alias of QuotientModuleElement

identity_hom()

Return the identity homomorphism on self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.identity_hom()
Matrix([
[1, 0], : QQ[x]**2/<[1, 2], [1, x]> -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

is_submodule(other)

Return True if other is a submodule of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> S = Q.submodule([1, 0])
>>> Q.is_submodule(S)
True
>>> S.is_submodule(Q)
False
```

is_zero()

Return True if self is a zero module.

This happens if and only if the base module is the same as the submodule being killed.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> (F/[(1, 0)]).is_zero()
False
>>> (F/[(1, 0), (0, 1)]).is_zero()
True
```

quotient_hom()

Return the quotient homomorphism to self.

That is, return a homomorphism representing the natural map from self.base to self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.quotient_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

submodule(*gens, **opts)

Generate a submodule.

This is the same as taking a quotient of a submodule of the base module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> Q.submodule([x, 0])
<[x, 0] + <[x, x]>>
```

class sympy.polys.agca.modules.SubQuotientModule(gens, container, **opts)

Submodule of a quotient module.

Equivalently, quotient module of a submodule.

Do not instantiate this, instead use the submodule or quotient_module constructing methods:

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S = F.submodule([1, 0], [1, x])
>>> Q = F/[(1, 0)]
>>> S/[(1, 0)] == Q.submodule([5, x])
True
```

Attributes:

- base - base module we are quotient of
- killed_module - submodule used to form the quotient

is_full_module()

Return True if self is the entire free module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
```

(continues on next page)

(continued from previous page)

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

quotient_hom()

Return the quotient homomorphism to self.

That is, return the natural map from `self.base` to `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = (QQ.old_poly_ring(x).free_module(2) / [(1, x)]).submodule([1, 0])
>>> M.quotient_hom()
Matrix([
[1, 0], : <[1, 0], [1, x]> -> <[1, 0] + <[1, x]>, [1, x] + <[1, x]>>
[0, 1]])
```

Module Homomorphisms and Syzygies

Let M and N be A -modules. A mapping $f : M \rightarrow N$ satisfying various obvious properties (see [Atiyah69] (page 1867)) is called an A -module homomorphism. In this case M is called the domain and N the codomain. The set $\{x \in M | f(x) = 0\}$ is called the kernel $\ker(f)$, whereas the set $\{f(x) | x \in M\}$ is called the image $\operatorname{im}(f)$. The kernel is a submodule of M , the image is a submodule of N . The homomorphism f is injective if and only if $\ker(f) = 0$ and surjective if and only if $\operatorname{im}(f) = N$. A bijective homomorphism is called an isomorphism. Equivalently, $\ker(f) = 0$ and $\operatorname{im}(f) = N$. (A related notion, which currently has no special name in the AGCA module, is that of the cokernel, $\operatorname{coker}(f) = N/\operatorname{im}(f)$.)

Suppose now M is an A -module. M is called finitely generated if there exists a surjective homomorphism $A^n \rightarrow M$ for some n . If such a morphism f is chosen, the images of the standard basis of A^n are called the generators of M . The module $\ker(f)$ is called syzygy module with respect to the generators. A module is called finitely presented if it is finitely generated with a finitely generated syzygy module. The class of finitely presented modules is essentially the largest class we can hope to be able to meaningfully compute in.

It is an important theorem that, for all the rings we are considering, all submodules of finitely generated modules are finitely generated, and hence finitely generated and finitely presented modules are the same.

The notion of syzygies, while it may first seem rather abstract, is actually very computational. This is because there exist (fairly easy) algorithms for computing them, and more general questions (kernels, intersections, ...) are often reduced to syzygy computation.

Let us say a few words about the definition of homomorphisms in the AGCA module. Suppose first that $f : M \rightarrow N$ is an arbitrary morphism of A -modules. Then if K is a submodule of M , f naturally defines a new homomorphism $g : K \rightarrow N$ (via $g(x) = f(x)$), called the restriction of f to K . If now K contained in the kernel of f , then moreover f defines in a natural homomorphism $g : M/K \rightarrow N$ (same formula as above!), and we say that f descends to M/K . Similarly, if L is a submodule of N , there is a natural homomorphism $g : M \rightarrow N/L$, we say that g factors through f . Finally, if now L contains the image of f , then there is a natural homomorphism $g : M \rightarrow L$ (defined, again, by the same formula), and we say g is obtained from f by restriction of codomain. Observe also that each of these four operations is reversible, in the sense that given g , one can always (non-uniquely) find f such that g is obtained from f in the above way.

Note that all modules implemented in AGCA are obtained from free modules by taking a succession of submodules and quotients. Hence, in order to explain how to define a homomorphism between arbitrary modules, in light of the above, we need only explain how to define homomorphisms of free modules. But, essentially by the definition of free module, a homomorphism from a free module A^n to any module M is precisely the same as giving n elements of M (the images of the standard basis), and giving an element of a free module A^m is precisely the same as giving m elements of A . Hence a homomorphism of free modules $A^n \rightarrow A^m$ can be specified via a matrix, entirely analogously to the case of vector spaces.

The functions `restrict_domain` etc. of the class `Homomorphism` can be used to carry out the operations described above, and homomorphisms of free modules can in principle be instantiated by hand. Since these operations are so common, there is a convenience function `homomorphism` to define a homomorphism between arbitrary modules via the method outlined above. It is essentially the only way homomorphisms need ever be created by the user.

`sympy.polys.agca.homomorphisms.homomorphism(domain, codomain, matrix)`

Create a homomorphism object.

This function tries to build a homomorphism from `domain` to `codomain` via the matrix `matrix`.

Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> R = QQ.old_poly_ring(x)
>>> T = R.free_module(2)
```

If `domain` is a free module generated by $e_1, \dots, e_n$, then `matrix` should be an n -element iterable $(b_1, \dots, b_n)$ where the b_i are elements of `codomain`. The constructed homomorphism is the unique homomorphism sending e_i to b_i .

```
>>> F = R.free_module(2)
>>> h = homomorphism(F, T, [[1, x], [x**2, 0]])
>>> h
Matrix([
[1, x**2], : QQ[x]**2 -> QQ[x]**2
[x,      0]])
>>> h([1, 0])
[1, x]
>>> h([0, 1])
[x**2, 0]
>>> h([1, 1])
[x**2 + 1, x]
```

If `domain` is a submodule of a free module, then `matrix` determines a homomorphism from the containing free module to `codomain`, and the homomorphism returned is obtained by restriction to `domain`.

```
>>> S = F.submodule([1, 0], [0, x])
>>> homomorphism(S, T, [[1, x], [x**2, 0]])
Matrix([
[1, x**2], : <[1, 0], [0, x]> -> QQ[x]**2
[x,      0]])
```

If domain is a (sub)quotient N/K , then `matrix` determines a homomorphism from N to codomain. If the kernel contains K , this homomorphism descends to domain and is returned; otherwise an exception is raised.

```
>>> homomorphism(S/[(1, 0)], T, [0, [x**2, 0]])
Matrix([
[0, x**2], : <[1, 0] + <[1, 0]>, [0, x] + <[1, 0]>, [1, 0] + <[1, 0]>> -> QQ[x]**2
[0, 0]])
>>> homomorphism(S/[(0, x)], T, [0, [x**2, 0]])
Traceback (most recent call last):
...
ValueError: kernel <[1, 0], [0, 0]> must contain sm, got <[0,x]>
```

Finally, here is the detailed reference of the actual homomorphism class:

class `sympy.polys.agca.homomorphisms.ModuleHomomorphism`(domain, codomain)
Abstract base class for module homomorphisms. Do not instantiate.

Instead, use the `homomorphism` function:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [0, 1]])
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

Attributes:

- `ring` - the ring over which we are considering modules
- `domain` - the domain module
- `codomain` - the codomain module
- `_ker` - cached kernel
- `_img` - cached image

Non-implemented methods:

- `_kernel`
- `_image`
- `_restrict_domain`
- `_restrict_codomain`
- `_quotient_domain`
- `_quotient_codomain`
- `_apply`
- `_mul_scalar`
- `_compose`
- `_add`

image()

Compute the image of `self`.

That is, if `self` is the homomorphism $\phi : M \rightarrow N$, then compute $\text{im}(\phi) = \{\phi(x) | x \in M\}$. This is a submodule of N .

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).image() == F.submodule([1, 0])
True
```

`is_injective()`

Return True if `self` is injective.

That is, check if the elements of the domain are mapped to the same codomain element.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_injective()
False
>>> h.quotient_domain(h.kernel()).is_injective()
True
```

`is_isomorphism()`

Return True if `self` is an isomorphism.

That is, check if every element of the codomain has precisely one preimage. Equivalently, `self` is both injective and surjective.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h = h.restrict_codomain(h.image())
>>> h.is_isomorphism()
False
>>> h.quotient_domain(h.kernel()).is_isomorphism()
True
```

`is_surjective()`

Return True if `self` is surjective.

That is, check if every element of the codomain has at least one preimage.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
```

(continues on next page)

(continued from previous page)

```
>>> h.is_surjective()
False
>>> h.restrict_codomain(h.image()).is_surjective()
True
```

is_zero()

Return True if self is a zero morphism.

That is, check if every element of the domain is mapped to zero under self.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_zero()
False
>>> h.restrict_domain(F.submodule()).is_zero()
True
>>> h.quotient_codomain(h.image()).is_zero()
True
```

kernel()

Compute the kernel of self.

That is, if self is the homomorphism $\phi : M \rightarrow N$, then compute $\ker(\phi) = \{x \in M | \phi(x) = 0\}$. This is a submodule of M .

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).kernel()
<[x, -1]>
```

quotient_codomain(sm)

Return self with codomain replaced by codomain/sm.

Here sm must be a submodule of self.codomain.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_codomain(F.submodule([1, 1]))
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

This is the same as composing with the quotient map on the left:

```
>>> (F/[(1, 1)]).quotient_hom() * h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

quotient_domain(sm)

Return self with domain replaced by domain/sm.

Here sm must be a submodule of self.kernel().

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_domain(F.submodule([-x, 1]))
Matrix([
[1, x], : QQ[x]**2/<[-x, 1]> -> QQ[x]**2
[0, 0]])
```

restrict_codomain(sm)

Return self, with codomain restricted to to sm.

Here sm has to be a submodule of self.codomain containing the image.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_codomain(F.submodule([1, 0]))
Matrix([
[1, x], : QQ[x]**2 -> <[1, 0]>
[0, 0]])
```

restrict_domain(sm)

Return self, with the domain restricted to sm.

Here sm has to be a submodule of self.domain.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
```

(continues on next page)

(continued from previous page)

```
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_domain(F.submodule([1, 0]))
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

This is the same as just composing on the right with the submodule inclusion:

```
>>> h * F.submodule([1, 0]).inclusion_hom()
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

Internals of the Polynomial Manipulation Module

The implementation of the polynomials module is structured internally in “levels”. There are four levels, called L0, L1, L2 and L3. The levels three and four contain the user-facing functionality and were described in the previous section. This section focuses on levels zero and one.

Level zero provides core polynomial manipulation functionality with C-like, low-level interfaces. Level one wraps this low-level functionality into object oriented structures. These are not the classes seen by the user, but rather classes used internally throughout the polys module.

There is one additional complication in the implementation. This comes from the fact that all polynomial manipulations are relative to a ground domain. For example, when factoring a polynomial like $x^{10} - 1$, one has to decide what ring the coefficients are supposed to belong to, or less trivially, what coefficients are allowed to appear in the factorization. This choice of coefficients is called a ground domain. Typical choices include the integers $\mathbb{Z}$, the rational numbers $\mathbb{Q}$ or various related rings and fields. But it is perfectly legitimate (although in this case uninteresting) to factorize over polynomial rings such as $k[Y]$, where k is some fixed field.

Thus the polynomial manipulation algorithms (both complicated ones like factoring, and simpler ones like addition or multiplication) have to rely on other code to manipulate the coefficients. In the polynomial manipulation module, such code is encapsulated in so-called “domains”. A domain is basically a factory object: it takes various representations of data, and converts them into objects with unified interface. Every object created by a domain has to implement the arithmetic operations $+$, $-$ and $\times$. Other operations are accessed through the domain, e.g. as in `ZZ.quo(ZZ(4), ZZ(2))`.

Note that there is some amount of circularity: the polynomial ring domains use the level one classes, the level one classes use the level zero functions, and level zero functions use domains. It is possible, in principle, but not in the current implementation, to work in rings like $k[X][Y]$. This would create even more layers. For this reason, working in the isomorphic ring $k[X, Y]$ is preferred.

Domains

Here we document the various implemented ground domains. There are three types: abstract domains, concrete domains, and “implementation domains”. Abstract domains cannot be

(usefully) instantiated at all, and just collect together functionality shared by many other domains. Concrete domains are those meant to be instantiated and used in the polynomial manipulation algorithms. In some cases, there are various possible ways to implement the data type the domain provides. For example, depending on what libraries are available on the system, the integers are implemented either using the python built-in integers, or using gmpy. Note that various aliases are created automatically depending on the libraries available. As such e.g. ZZ always refers to the most efficient implementation of the integer ring available.

Abstract Domains

class sympy.polys.domains.domain.Domain

Represents an abstract domain.

Attributes

alias	
dtype	
one	
rep	
zero	

abs(a)

Absolute value of a, implies `__abs__`.

add(a, b)

Sum of a and b, implies `__add__`.

algebraic_field(*extension)

Returns an algebraic field, i.e. $K(\alpha, \dots)$.

almosteq(a, b, tolerance=None)

Check if a and b are almost equal.

characteristic()

Return the characteristic of this domain.

cofactors(a, b)

Returns GCD and cofactors of a and b.

convert(element, base=None)

Convert element to `self.dtype`.

convert_from(element, base)

Convert element to `self.dtype` given the base domain.

denom(a)

Returns denominator of a.

div(a, b)

Division of a and b, implies something.

evalf(a, prec=None, **options)

Returns numerical approximation of a.

exquo(a, b)

Exact quotient of a and b, implies something.

frac_field(*symbols, **kwargs)
Returns a fraction field, i.e. $K(X)$.

from_AlgebraicField(a, K0)
Convert an algebraic number to dtype.

from_ComplexField(a, K0)
Convert a complex element to dtype.

from_ExpressionDomain(a, K0)
Convert a EX object to dtype.

from_FF_gmpy(a, K0)
Convert ModularInteger(mpz) to dtype.

from_FF_python(a, K0)
Convert ModularInteger(int) to dtype.

from_FractionField(a, K0)
Convert a rational function to dtype.

from_GlobalPolynomialRing(a, K0)
Convert a polynomial to dtype.

from_PolynomialRing(a, K0)
Convert a polynomial to dtype.

from_QQ_gmpy(a, K0)
Convert a GMPY mpq object to dtype.

from_QQ_python(a, K0)
Convert a Python Fraction object to dtype.

from_RealField(a, K0)
Convert a real element object to dtype.

from_ZZ_gmpy(a, K0)
Convert a GMPY mpz object to dtype.

from_ZZ_python(a, K0)
Convert a Python int object to dtype.

from_sympy(a)
Convert a SymPy object to dtype.

gcd(a, b)
Returns GCD of a and b.

gcdex(a, b)
Extended GCD of a and b.

get_exact()
Returns an exact domain associated with self.

get_field()
Returns a field associated with self.

get_ring()
Returns a ring associated with self.

half_gcdex(a, b)
Half extended GCD of a and b.

inject(*symbols)
Inject generators into this domain.

invert(a, b)
Returns inversion of $a \bmod b$, implies something.

is_negative(a)
Returns True if a is negative.

is_nonnegative(a)
Returns True if a is non-negative.

is_nonpositive(a)
Returns True if a is non-positive.

is_one(a)
Returns True if a is one.

is_positive(a)
Returns True if a is positive.

is_zero(a)
Returns True if a is zero.

lcm(a, b)
Returns LCM of a and b.

log(a, b)
Returns b-base logarithm of a.

map(seq)
Rersively apply self to all elements of seq.

mul(a, b)
Product of a and b, implies `__mul__`.

n(a, prec=None, **options)
Returns numerical approximation of a.

neg(a)
Returns a negated, implies `__neg__`.

numer(a)
Returns numerator of a.

of_type(element)
Check if a is of type dtype.

old_frac_field(*symbols, **kwargs)
Returns a fraction field, i.e. $K(X)$.

old_poly_ring(*symbols, **kwargs)
Returns a polynomial ring, i.e. $K[X]$.

poly_ring(*symbols, **kwargs)
Returns a polynomial ring, i.e. $K[X]$.

pos(a)
Returns a positive, implies `__pos__`.

pow(a, b)
Raise a to power b, implies `__pow__`.

quo(a, b)
Quotient of a and b, implies something.

rem(a, b)
Remainder of a and b, implies `__mod__`.

revert(a)

Returns $a^{**(-1)}$ if possible.

sqrt(a)

Returns square root of a.

sub(a, b)

Difference of a and b, implies `__sub__`.

to_sympy(a)

Convert a to a SymPy object.

unify(K1, symbols=None)

Construct a minimal domain that contains elements of K0 and K1.

Known domains (from smallest to largest):

- GF(p)
- ZZ
- QQ
- RR(prec, tol)
- CC(prec, tol)
- ALG(a, b, c)
- K[x, y, z]
- K(x, y, z)
- EX

class sympy.polys.domains.field.**Field**

Represents a field domain.

Attributes

alias	
dtype	
one	
rep	
zero	

div(a, b)

Division of a and b, implies `__div__`.

exquo(a, b)

Exact quotient of a and b, implies `__div__`.

gcd(a, b)

Returns GCD of a and b.

This definition of GCD over fields allows to clear denominators in *primitive()*.

```
>>> from sympy.polys.domains import QQ
>>> from sympy import S, gcd, primitive
>>> from sympy.abc import x
```

```
>>> QQ.gcd(QQ(2, 3), QQ(4, 9))
2/9
>>> gcd(S(2)/3, S(4)/9)
2/9
>>> primitive(2*x/3 + S(4)/9)
(2/9, 3*x + 2)
```

get_field()

Returns a field associated with self.

get_ring()

Returns a ring associated with self.

lcm(a, b)

Returns LCM of a and b.

```
>>> from sympy.polys.domains import QQ
>>> from sympy import S, lcm
```

```
>>> QQ.lcm(QQ(2, 3), QQ(4, 9))
4/3
>>> lcm(S(2)/3, S(4)/9)
4/3
```

quo(a, b)

Quotient of a and b, implies `__div__`.

rem(a, b)

Remainder of a and b, implies nothing.

revert(a)

Returns $a^{**(-1)}$ if possible.

class sympy.polys.domains.ring.Ring

Represents a ring domain.

Attributes

alias	
dtype	
one	
rep	
zero	

denom(a)

Returns denominator of a .

div(a, b)

Division of a and b, implies `__divmod__`.

exquo(a, b)

Exact quotient of a and b, implies `__floordiv__`.

free_module(rank)

Generate a free module of rank rank over self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

get_ring()

Returns a ring associated with self.

ideal(*gens)

Generate an ideal of self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

invert(a, b)

Returns inversion of $a \bmod b$.

numer(a)

Returns numerator of a .

quo(a, b)

Quotient of a and b , implies `__floordiv__`.

quotient_ring(e)

Form a quotient ring of self.

Here e can be an ideal or an iterable.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

rem(a, b)

Remainder of a and b , implies `__mod__`.

revert(a)

Returns $a^{**(-1)}$ if possible.

class sympy.polys.domains.simpdomain.SimpleDomain

Base class for simple domains, e.g. $\mathbb{ZZ}$, $\mathbb{QQ}$.

Attributes

alias	
dtype	
one	
rep	
zero	

inject(*gens)

Inject generators into this domain.

class sympy.polys.domains.compositedomain.**CompositeDomain**

Base class for composite domains, e.g. $\mathbb{ZZ}[x]$, $\mathbb{ZZ}(X)$.

Attributes

alias	
domain	
dtype	
gens	
ngens	
one	
rep	
symbols	
zero	

inject(*symbols)

Inject generators into this domain.

Concrete Domains

class sympy.polys.domains.**FiniteField**(mod, dom=None, symmetric=True)

General class for finite fields.

Attributes

alias	
dom	
dtype	
mod	
one	
zero	

characteristic()

Return the characteristic of this domain.

from_FF_gmpy(a, K0=None)

Convert ModularInteger(mpz) to dtype.

from_FF_python(a, K0=None)

Convert ModularInteger(int) to dtype.

from_QQ_gmpy(a, K0=None)

Convert GMPY's mpq to dtype.

from_QQ_python(a, K0=None)

Convert Python's Fraction to dtype.

from_RealField(a, K0)

Convert mpmath's mpf to dtype.

from_ZZ_gmpy(a, K0=None)
Convert GMPY's mpz to dtype.

from_ZZ_python(a, K0=None)
Convert Python's int to dtype.

from_sympy(a)
Convert SymPy's Integer to SymPy's Integer.

get_field()
Returns a field associated with self.

to_sympy(a)
Convert a to a SymPy object.

class sympy.polys.domains.**IntegerRing**
General class for integer rings.

Attributes

alias	
dtype	
one	
zero	

algebraic_field(*extension)
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

from_AlgebraicField(a, K0)
Convert a ANP object to dtype.

get_field()
Returns a field associated with self.

log(a, b)
Returns b-base logarithm of a.

class sympy.polys.domains.**PolynomialRing**(domain_or_ring, symbols=None, order=None)
A class for representing multivariate polynomial rings.

Attributes

alias	
domain	
dtype	
gens	
ngens	
rep	
symbols	

factorial(a)
Returns factorial of a .

from_AlgebraicField(a, K0)
Convert an algebraic number to dtype.

```

from_FractionField(a, K0)
    Convert a rational function to dtype.
from_PolynomialRing(a, K0)
    Convert a polynomial to dtype.
from_QQ_gmpy(a, K0)
    Convert a GMPY mpq object to dtype.
from_QQ_python(a, K0)
    Convert a Python Fraction object to dtype.
from_RealField(a, K0)
    Convert a mpmath mpf object to dtype.
from_ZZ_gmpy(a, K0)
    Convert a GMPY mpz object to dtype.
from_ZZ_python(a, K0)
    Convert a Python int object to dtype.
from_sympy(a)
    Convert SymPy's expression to dtype.
gcd(a, b)
    Returns GCD of  $a$  and  $b$ .
gcdex(a, b)
    Extended GCD of  $a$  and  $b$ .
get_field()
    Returns a field associated with self.
is_negative(a)
    Returns True if  $LC(a)$  is negative.
is_nonnegative(a)
    Returns True if  $LC(a)$  is non-negative.
is_nonpositive(a)
    Returns True if  $LC(a)$  is non-positive.
is_positive(a)
    Returns True if  $LC(a)$  is positive.
lcm(a, b)
    Returns LCM of  $a$  and  $b$ .
to_sympy(a)
    Convert  $a$  to a SymPy object.
class sympy.polys.domains.RationalField
    General class for rational fields.

```

Attributes

alias	
dtype	
one	
zero	

algebraic_field(*extension)
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

from_AlgebraicField(a, K0)
Convert a ANP object to dtype.

class sympy.polys.domains.**AlgebraicField**(dom, *ext)
A class for representing algebraic number fields.

Attributes

alias	
one	
rep	
zero	

algebraic_field(*extension)
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

denom(a)
Returns denominator of a.

dtype
alias of [sympy.polys.polyclasses.ANP](#) (page 924)

from_QQ_gmpy(a, K0)
Convert a GMPY mpq object to dtype.

from_QQ_python(a, K0)
Convert a Python Fraction object to dtype.

from_RealField(a, K0)
Convert a mpmath mpf object to dtype.

from_ZZ_gmpy(a, K0)
Convert a GMPY mpz object to dtype.

from_ZZ_python(a, K0)
Convert a Python int object to dtype.

from_sympy(a)
Convert SymPy's expression to dtype.

get_ring()
Returns a ring associated with self.

is_negative(a)
Returns True if a is negative.

is_nonnegative(a)
Returns True if a is non-negative.

is_nonpositive(a)
Returns True if a is non-positive.

is_positive(a)
Returns True if a is positive.

numer(a)
Returns numerator of a.

to_sympy(a)
Convert a to a SymPy object.

class sympy.polys.domains.**FractionField**(domain_or_field, symbols=None, order=None)
A class for representing multivariate rational function fields.

Attributes

alias	
domain	
dtype	
gens	
ngens	
rep	
symbols	

denom(a)
Returns denominator of *a*.

factorial(a)
Returns factorial of *a*.

from_AlgebraicField(a, K0)
Convert an algebraic number to *dtype*.

from_FractionField(a, K0)
Convert a rational function to *dtype*.

from_PolynomialRing(a, K0)
Convert a polynomial to *dtype*.

from_QQ_gmpy(a, K0)
Convert a GMPY *mpq* object to *dtype*.

from_QQ_python(a, K0)
Convert a Python *Fraction* object to *dtype*.

from_RealField(a, K0)
Convert a mpmath *mpf* object to *dtype*.

from_ZZ_gmpy(a, K0)
Convert a GMPY *mpz* object to *dtype*.

from_ZZ_python(a, K0)
Convert a Python *int* object to *dtype*.

from_sympy(a)
Convert SymPy's expression to *dtype*.

get_ring()
Returns a field associated with *self*.

is_negative(a)
Returns True if $LC(a)$ is negative.

is_nonnegative(a)
Returns True if $LC(a)$ is non-negative.

is_nonpositive(a)
Returns True if $LC(a)$ is non-positive.

is_positive(a)

Returns True if $LC(a)$ is positive.

numer(a)

Returns numerator of a.

to_sympy(a)

Convert a to a SymPy object.

class sympy.polys.domains.**RealField**(prec=53, dps=None, tol=None)

Real numbers up to the given precision.

Attributes

alias	
dtype	
one	
zero	

almosteq(a, b, tolerance=None)

Check if a and b are almost equal.

from_sympy(expr)

Convert SymPy's number to dtype.

gcd(a, b)

Returns GCD of a and b.

get_exact()

Returns an exact domain associated with self.

get_ring()

Returns a ring associated with self.

lcm(a, b)

Returns LCM of a and b.

to_rational(element, limit=True)

Convert a real number to rational number.

to_sympy(element)

Convert element to SymPy number.

class sympy.polys.domains.**ExpressionDomain**

A class for arbitrary expressions.

Attributes

alias	

class **Expression**(ex)

An arbitrary expression.

denom(a)

Returns denominator of a.

dtype

alias of [ExpressionDomain.Expression](#) (page 916)

```

from_ExpressionDomain(a, K0)
    Convert a EX object to dtype.
from_FractionField(a, K0)
    Convert a DMF object to dtype.
from_PolynomialRing(a, K0)
    Convert a DMP object to dtype.
from_QQ_gmpy(a, K0)
    Convert a GMPY mpq object to dtype.
from_QQ_python(a, K0)
    Convert a Python Fraction object to dtype.
from_RealField(a, K0)
    Convert a mpmath mpf object to dtype.
from_ZZ_gmpy(a, K0)
    Convert a GMPY mpz object to dtype.
from_ZZ_python(a, K0)
    Convert a Python int object to dtype.
from_sympy(a)
    Convert SymPy's expression to dtype.
get_field()
    Returns a field associated with self.
get_ring()
    Returns a ring associated with self.
is_negative(a)
    Returns True if a is negative.
is_nonnegative(a)
    Returns True if a is non-negative.
is_nonpositive(a)
    Returns True if a is non-positive.
is_positive(a)
    Returns True if a is positive.
numer(a)
    Returns numerator of a.
to_sympy(a)
    Convert a to a SymPy object.

```

Implementation Domains

```

class sympy.polys.domains.PythonFiniteField(mod, symmetric=True)
    Finite field based on Python's integers.

```

Attributes

dom	
dtype	
mod	
one	
zero	

class `sympy.polys.domains.GMPYFiniteField(mod, symmetric=True)`
Finite field based on GMPY integers.

Attributes

dom	
dtype	
mod	
one	
zero	

class `sympy.polys.domains.PythonIntegerRing`
Integer ring based on Python's int type.

class `sympy.polys.domains.GMPYIntegerRing`
Integer ring based on GMPY's mpz type.

class `sympy.polys.domains.PythonRationalField`
Rational field based on Python rational number type.

class `sympy.polys.domains.GMPYRationalField`
Rational field based on GMPY mpq class.

Level One

class `sympy.polys.polyclasses.DMP(rep, dom, lev=None, ring=None)`
Dense Multivariate Polynomials over K .

LC()
Returns the leading coefficient of f .

TC()
Returns the trailing coefficient of f .

abs()
Make all coefficients in f positive.

add(g)
Add two multivariate polynomials f and g .

add_ground(c)
Add an element of the ground domain to f .

all_coeffs()
Returns all coefficients from f .

all_monoms()
Returns all monomials from f .

all_terms()
Returns all terms from a f .

cancel(g , include=True)
Cancel common factors in a rational function f/g .

clear_denoms()
Clear denominators, but keep the ground domain.

coeffs(order=None)
Returns all non-zero coefficients from f in lex order.

cofactors(g)
Returns GCD of f and g and their cofactors.

compose(g)
Computes functional composition of f and g .

content()
Returns GCD of polynomial coefficients.

convert(dom)
Convert the ground domain of f .

count_complex_roots(inf=None, sup=None)
Return the number of complex roots of f in $[inf, sup]$.

count_real_roots(inf=None, sup=None)
Return the number of real roots of f in $[inf, sup]$.

decompose()
Computes functional decomposition of f .

deflate()
Reduce degree of f by mapping x_i^m to y_i .

degree($j=0$)
Returns the leading degree of f in x_j .

degree_list()
Returns a list of degrees of f .

diff($m=1, j=0$)
Computes the m -th order derivative of f in x_j .

discriminant()
Computes discriminant of f .

div(g)
Polynomial division with remainder of f and g .

eject(dom, front=False)
Eject selected generators into the ground domain.

eval($a, j=0$)
Evaluates f at the given point a in x_j .

exclude()
Remove useless generators from f .
Returns the removed generators and the new excluded f .

Examples

```
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.domains import ZZ
```

```
>>> DMP([[[ZZ(1)]], [[ZZ(1)], [ZZ(2)]]], ZZ).exclude()
([2], DMP([[1], [1, 2]], ZZ, None))
```

exquo(g)

Computes polynomial exact quotient of f and g.

exquo_ground(c)

Exact quotient of f by a an element of the ground domain.

factor_list()

Returns a list of irreducible factors of f.

factor_list_include()

Returns a list of irreducible factors of f.

classmethod from_dict(rep, lev, dom)

Construct and instance of cls from a dict representation.

classmethod from_list(rep, lev, dom)

Create an instance of cls given a list of native coefficients.

classmethod from_sympy_list(rep, lev, dom)

Create an instance of cls given a list of SymPy coefficients.

gcd(g)

Returns polynomial GCD of f and g.

gcdex(g)

Extended Euclidean algorithm, if univariate.

gff_list()

Computes greatest factorial factorization of f.

half_gcdex(g)

Half extended Euclidean algorithm, if univariate.

homogeneous_order()

Returns the homogeneous order of f.

homogenize(s)

Return homogeneous polynomial of f

inject(front=False)

Inject ground domain generators into f.

integrate(m=1, j=0)

Computes the m-th order indefinite integral of f in x_j.

intervals(all=False, eps=None, inf=None, sup=None, fast=False, sqf=False)

Compute isolating intervals for roots of f.

invert(g)

Invert f modulo g, if possible.

is_cyclotomic

Returns True if f is a cyclotomic polynomial.

is_ground

Returns True if f is an element of the ground domain.

is_homogeneous
Returns True if f is a homogeneous polynomial.

is_irreducible
Returns True if f has no factors over its domain.

is_linear
Returns True if f is linear in all its variables.

is_monic
Returns True if the leading coefficient of f is one.

is_monomial
Returns True if f is zero or has only one term.

is_one
Returns True if f is a unit polynomial.

is_primitive
Returns True if the GCD of the coefficients of f is one.

is_quadratic
Returns True if f is quadratic in all its variables.

is_sqf
Returns True if f is a square-free polynomial.

is_zero
Returns True if f is a zero polynomial.

l1_norm()
Returns l1 norm of f .

lcm(g)
Returns polynomial LCM of f and g .

lift()
Convert algebraic coefficients to rationals.

max_norm()
Returns maximum norm of f .

monic()
Divides all coefficients by $LC(f)$.

monoms(order=None)
Returns all non-zero monomials from f in lex order.

mul(g)
Multiply two multivariate polynomials f and g .

mul_ground(c)
Multiply f by a an element of the ground domain.

neg()
Negate all coefficients in f .

norm()
Computes $\text{Norm}(f)$.

nth(*N)
Returns the n -th coefficient of f .

pdiv(g)
Polynomial pseudo-division of f and g .

per(rep, dom=None, kill=False, ring=None)
Create a DMP out of the given representation.

permute(P)
Returns a polynomial in $K[x_{P(1)}, \dots, x_{P(n)}]$.

Examples

```
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.domains import ZZ
```

```
>>> DMP([[[ZZ(2)], [ZZ(1), ZZ(0)]]], [[]], ZZ).permute([1, 0, 2])
DMP([[[2], []], [[1, 0], []]], ZZ, None)
```

```
>>> DMP([[[ZZ(2)], [ZZ(1), ZZ(0)]]], [[]], ZZ).permute([1, 2, 0])
DMP([[[1], []], [[2, 0], []]], ZZ, None)
```

pexquo(g)
Polynomial exact pseudo-quotient of f and g.

pow(n)
Raise f to a non-negative power n.

pquo(g)
Polynomial pseudo-quotient of f and g.

prem(g)
Polynomial pseudo-remainder of f and g.

primitive()
Returns content and a primitive form of f.

quo(g)
Computes polynomial quotient of f and g.

quo_ground(c)
Quotient of f by a an element of the ground domain.

refine_root(s, t, eps=None, steps=None, fast=False)
Refine an isolating interval to the given precision.
eps should be a rational number.

rem(g)
Computes polynomial remainder of f and g.

resultant(g, includePRS=False)
Computes resultant of f and g via PRS.

revert(n)
Compute $f^{*(-1)} \bmod x^{*n}$.

shift(a)
Efficiently compute Taylor shift $f(x + a)$.

slice(m, n, j=0)
Take a continuous subsequence of terms of f.

sqf_list(all=False)
Returns a list of square-free factors of f.

sqf_list_include(all=False)
Returns a list of square-free factors of f .

sqf_norm()
Computes square-free norm of f .

sqf_part()
Computes square-free part of f .

sqr()
Square a multivariate polynomial f .

sturm()
Computes the Sturm sequence of f .

sub(g)
Subtract two multivariate polynomials f and g .

sub_ground(c)
Subtract an element of the ground domain from f .

subresultants(g)
Computes subresultant PRS sequence of f and g .

terms(order=None)
Returns all non-zero terms from f in lex order.

terms_gcd()
Remove GCD of terms from the polynomial f .

to_dict(zero=False)
Convert f to a dict representation with native coefficients.

to_exact()
Make the ground domain exact.

to_field()
Make the ground domain a field.

to_list()
Convert f to a list representation with native coefficients.

to_ring()
Make the ground domain a ring.

to_sympy_dict(zero=False)
Convert f to a dict representation with SymPy coefficients.

to_sympy_list()
Convert f to a list representation with SymPy coefficients.

to_tuple()
Convert f to a tuple representation with native coefficients.
This is needed for hashing.

total_degree()
Returns the total degree of f .

transform(p, q)
Evaluate functional transformation $q^{*n} * f(p/q)$.

trunc(p)
Reduce f modulo a constant p .

unify(g)
Unify representations of two multivariate polynomials.

class sympy.polys.polyclasses.**DMF**(rep, dom, lev=None, ring=None)
Dense Multivariate Fractions over K .

add(g)
Add two multivariate fractions f and g .

cancel()
Remove common factors from f .num and f .den.

denom()
Returns the denominator of f .

exquo(g)
Computes quotient of fractions f and g .

frac_unify(g)
Unify representations of two multivariate fractions.

half_per(rep, kill=False)
Create a DMP out of the given representation.

invert(check=True)
Computes inverse of a fraction f .

is_one
Returns True if f is a unit fraction.

is_zero
Returns True if f is a zero fraction.

mul(g)
Multiply two multivariate fractions f and g .

neg()
Negate all coefficients in f .

numer()
Returns the numerator of f .

per(num, den, cancel=True, kill=False, ring=None)
Create a DMF out of the given representation.

poly_unify(g)
Unify a multivariate fraction and a polynomial.

pow(n)
Raise f to a non-negative power n .

quo(g)
Computes quotient of fractions f and g .

sub(g)
Subtract two multivariate fractions f and g .

class sympy.polys.polyclasses.**ANP**(rep, mod, dom)
Dense Algebraic Number Polynomials over a field.

LC()
Returns the leading coefficient of f .

TC()
Returns the trailing coefficient of f .

is_ground
Returns True if f is an element of the ground domain.

is_one
Returns True if f is a unit algebraic number.

is_zero
Returns True if f is a zero algebraic number.

pow(n)
Raise f to a non-negative power n .

to_dict()
Convert f to a dict representation with native coefficients.

to_list()
Convert f to a list representation with native coefficients.

to_sympy_dict()
Convert f to a dict representation with SymPy coefficients.

to_sympy_list()
Convert f to a list representation with SymPy coefficients.

to_tuple()
Convert f to a tuple representation with native coefficients.
This is needed for hashing.

unify(g)
Unify representations of two algebraic numbers.

Level Zero

Level zero contains the bulk code of the polynomial manipulation module.

Manipulation of dense, multivariate polynomials

These functions can be used to manipulate polynomials in $K[X_0, \dots, X_u]$. Functions for manipulating multivariate polynomials in the dense representation have the prefix `dmp_`. Functions which only apply to univariate polynomials (i.e. $u = 0$) have the prefix `dup_`. The ground domain K has to be passed explicitly. For many multivariate polynomial manipulation functions also the level u , i.e. the number of generators minus one, has to be passed. (Note that, in many cases, `dup_` versions of functions are available, which may be slightly more efficient.)

Basic manipulation:

`sympy.polys.densebasic.dmp_LC(f, K)`
Return leading coefficient of f .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_LC
```

```
>>> poly_LC([], ZZ)
0
>>> poly_LC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
1
```

`sympy.polys.densebasic.dmp_TC(f, K)`
Return trailing coefficient of f .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_TC
```

```
>>> poly_TC([], ZZ)
0
>>> poly_TC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
3
```

`sympy.polys.densebasic.dmp_ground_LC(f, u, K)`
Return the ground leading coefficient.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_LC
```

```
>>> f = ZZ.map([[[1], [2, 3]]])
```

```
>>> dmp_ground_LC(f, 2, ZZ)
1
```

`sympy.polys.densebasic.dmp_ground_TC(f, u, K)`
Return the ground trailing coefficient.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_TC
```

```
>>> f = ZZ.map([[[1], [2, 3]]])
```

```
>>> dmp_ground_TC(f, 2, ZZ)
3
```

`sympy.polys.densebasic.dmp_true_LT(f, u, K)`
Return the leading term $c * x_1^{n_1} \dots x_k^{n_k}$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_true_LT
```

```
>>> f = ZZ.map([[4], [2, 0], [3, 0, 0]])
```

```
>>> dmp_true_LT(f, 1, ZZ)
((2, 0), 4)
```

`sympy.polys.densebasic.dmp_degree(f, u)`

Return the leading degree of f in x_u in $K[X]$.

Note that the degree of 0 is negative infinity (the SymPy object `-oo`).

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree
```

```
>>> dmp_degree([[]], 2)
-oo
```

```
>>> f = ZZ.map([[2], [1, 2, 3]])
```

```
>>> dmp_degree(f, 1)
1
```

`sympy.polys.densebasic.dmp_degree_in(f, j, u)`

Return the leading degree of f in x_j in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_in
```

```
>>> f = ZZ.map([[2], [1, 2, 3]])
```

```
>>> dmp_degree_in(f, 0, 1)
1
>>> dmp_degree_in(f, 1, 1)
2
```

`sympy.polys.densebasic.dmp_degree_list(f, u)`

Return a list of degrees of f in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_list
```

```
>>> f = ZZ.map([[1], [1, 2, 3]])
```

```
>>> dmp_degree_list(f, 1)
(1, 2)
```

`sympy.polys.densebasic.dmp_strip(f, u)`
Remove leading zeros from f in $K[X]$.

Examples

```
>>> from sympy.polys.densebasic import dmp_strip
```

```
>>> dmp_strip([], [0, 1, 2], [1]), 1)
[[0, 1, 2], [1]]
```

`sympy.polys.densebasic.dmp_validate(f, K=None)`
Return the number of levels in f and recursively strip it.

Examples

```
>>> from sympy.polys.densebasic import dmp_validate
```

```
>>> dmp_validate([], [0, 1, 2], [1])
([[1, 2], [1]], 1)
```

```
>>> dmp_validate([1], 1)
Traceback (most recent call last):
...
ValueError: invalid data structure for a multivariate polynomial
```

`sympy.polys.densebasic.dup_reverse(f)`
Compute $x^n \cdot f(1/x)$, i.e.: reverse f in $K[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_reverse
```

```
>>> f = ZZ.map([1, 2, 3, 0])
```

```
>>> dup_reverse(f)
[3, 2, 1]
```

`sympy.polys.densebasic.dmp_copy(f, u)`
Create a new copy of a polynomial f in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_copy
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_copy(f, 1)
[[1], [1, 2]]
```

`sympy.polys.densebasic.dmp_to_tuple(f, u)`

Convert f into a nested tuple of tuples.

This is needed for hashing. This is similar to `dmp_copy()`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_to_tuple
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_to_tuple(f, 1)
((1,), (1, 2))
```

`sympy.polys.densebasic.dmp_normal(f, u, K)`

Normalize a multivariate polynomial in the given domain.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_normal
```

```
>>> dmp_normal([], [0, 1.5, 2], 1, ZZ)
[[1, 2]]
```

`sympy.polys.densebasic.dmp_convert(f, u, K0, K1)`

Convert the ground domain of f from $K0$ to $K1$.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_convert
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> dmp_convert([[R(1)], [R(2)]], 1, R.to_domain(), ZZ)
[[1], [2]]
>>> dmp_convert([[ZZ(1)], [ZZ(2)]], 1, ZZ, R.to_domain())
[[1], [2]]
```

`sympy.polys.densebasic.dmp_from_sympy(f, u, K)`

Convert the ground domain of f from SymPy to K .

Examples

```
>>> from sympy import S
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_sympy
```

```
>>> dmp_from_sympy([[S(1)], [S(2)]], 1, ZZ) == [[ZZ(1)], [ZZ(2)]]
True
```

`sympy.polys.densebasic.dmp_nth(f, n, u, K)`
Return the n -th coefficient of f in $K[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nth
```

```
>>> f = ZZ.map([[1], [2], [3]])
```

```
>>> dmp_nth(f, 0, 1, ZZ)
[3]
>>> dmp_nth(f, 4, 1, ZZ)
[]
```

`sympy.polys.densebasic.dmp_ground_nth(f, N, u, K)`
Return the ground n -th coefficient of f in $K[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_nth
```

```
>>> f = ZZ.map([[1], [2, 3]])
```

```
>>> dmp_ground_nth(f, (0, 1), 1, ZZ)
2
```

`sympy.polys.densebasic.dmp_zero_p(f, u)`
Return True if f is zero in $K[X]$.

Examples

```
>>> from sympy.polys.densebasic import dmp_zero_p
```

```
>>> dmp_zero_p([[[[[]]]]], 4)
True
>>> dmp_zero_p([[[[[]]]], 4)
False
```


`sympy.polys.densebasic.dmp_zero(u)`
Return a multivariate zero.

Examples

```
>>> from sympy.polys.densebasic import dmp_zero
```

```
>>> dmp_zero(4)
[[[[]]]]
```

`sympy.polys.densebasic.dmp_one_p(f, u, K)`
Return True if f is one in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one_p
```

```
>>> dmp_one_p([[ZZ(1)]], 2, ZZ)
True
```

`sympy.polys.densebasic.dmp_one(u, K)`
Return a multivariate one over K .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one
```

```
>>> dmp_one(2, ZZ)
[[[1]]]
```

`sympy.polys.densebasic.dmp_ground_p(f, c, u)`
Return True if f is constant in $K[X]$.

Examples

```
>>> from sympy.polys.densebasic import dmp_ground_p
```

```
>>> dmp_ground_p([[3]], 3, 2)
True
>>> dmp_ground_p([[4]], None, 2)
True
```

`sympy.polys.densebasic.dmp_ground(c, u)`
Return a multivariate constant.

Examples

```
>>> from sympy.polys.densebasic import dmp_ground
```

```
>>> dmp_ground(3, 5)
[[[[[3]]]]]]
>>> dmp_ground(1, -1)
1
```

`sympy.polys.densebasic.dmp_zeros(n, u, K)`
Return a list of multivariate zeros.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_zeros
```

```
>>> dmp_zeros(3, 2, ZZ)
[[[]], [[]], [[]]]
>>> dmp_zeros(3, -1, ZZ)
[0, 0, 0]
```

`sympy.polys.densebasic.dmp_grounds(c, n, u)`
Return a list of multivariate constants.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_grounds
```

```
>>> dmp_grounds(ZZ(4), 3, 2)
[[[4]], [[4]], [[4]]]
>>> dmp_grounds(ZZ(4), 3, -1)
[4, 4, 4]
```

`sympy.polys.densebasic.dmp_negative_p(f, u, K)`
Return True if $LC(f)$ is negative.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_negative_p
```

```
>>> dmp_negative_p([ZZ(1)], [-ZZ(1)]), 1, ZZ)
False
>>> dmp_negative_p([-ZZ(1)], [ZZ(1)]), 1, ZZ)
True
```

`sympy.polys.densebasic.dmp_positive_p(f, u, K)`
Return True if $LC(f)$ is positive.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_positive_p
```

```
>>> dmp_positive_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
True
>>> dmp_positive_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
False
```

`sympy.polys.densebasic.dmp_from_dict(f, u, K)`
Create a $K[X]$ polynomial from a dict.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_dict
```

```
>>> dmp_from_dict({(0, 0): ZZ(3), (0, 1): ZZ(2), (2, 1): ZZ(1)}, 1, ZZ)
[[1, 0], [], [2, 3]]
>>> dmp_from_dict({}, 0, ZZ)
[]
```

`sympy.polys.densebasic.dmp_to_dict(f, u, K=None, zero=False)`
Convert a $K[X]$ polynomial to a dict ''.

Examples

```
>>> from sympy.polys.densebasic import dmp_to_dict
```

```
>>> dmp_to_dict([[1, 0], [], [2, 3]], 1)
{(0, 0): 3, (0, 1): 2, (2, 1): 1}
>>> dmp_to_dict([], 0)
{}
```

`sympy.polys.densebasic.dmp_swap(f, i, j, u, K)`
Transform $K[\dots x_i \dots x_j \dots]$ to $K[\dots x_j \dots x_i \dots]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_swap
```

```
>>> f = ZZ.map([[[2], [1, 0]], []])
```

```
>>> dmp_swap(f, 0, 1, 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_swap(f, 1, 2, 2, ZZ)
[[[1], [2, 0]], []]
```

(continues on next page)

(continued from previous page)

```
>>> dmp_swap(f, 0, 2, 2, ZZ)
[[[1, 0]], [[2, 0], []]]
```

`sympy.polys.densebasic.dmp_permute(f, P, u, K)`
Return a polynomial in $K[x_{P(1)}, \dots, x_{P(n)}]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_permute
```

```
>>> f = ZZ.map([[2], [1, 0]], [])
```

```
>>> dmp_permute(f, [1, 0, 2], 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_permute(f, [1, 2, 0], 2, ZZ)
[[[1], []], [[2, 0], []]]
```

`sympy.polys.densebasic.dmp_nest(f, l, K)`
Return a multivariate value nested l -levels.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nest
```

```
>>> dmp_nest([ZZ(1)], 2, ZZ)
[[[1]]]
```

`sympy.polys.densebasic.dmp_raise(f, l, u, K)`
Return a multivariate polynomial raised l -levels.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_raise
```

```
>>> f = ZZ.map([], [1, 2])
```

```
>>> dmp_raise(f, 2, 1, ZZ)
[[[]], [[1]], [[2]]]
```

`sympy.polys.densebasic.dmp_deflate(f, u, K)`
Map $x_i^{m_i}$ to y_i in a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_deflate
```

```
>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
```

```
>>> dmp_deflate(f, 1, ZZ)
((2, 3), [[1, 2], [3, 4]])
```

`sympy.polys.densebasic.dmp_multi_deflate(polys, u, K)`
Map $x_i^{m_i}$ to y_i in a set of polynomials in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_multi_deflate
```

```
>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
>>> g = ZZ.map([[1, 0, 2], [], [3, 0, 4]])
```

```
>>> dmp_multi_deflate((f, g), 1, ZZ)
((2, 1), ([[1, 0, 0, 2], [3, 0, 0, 4]], [[1, 0, 2], [3, 0, 4]]))
```

`sympy.polys.densebasic.dmp_inflate(f, M, u, K)`
Map y_i to $x_i^{k_i}$ in a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inflate
```

```
>>> f = ZZ.map([[1, 2], [3, 4]])
```

```
>>> dmp_inflate(f, (2, 3), 1, ZZ)
[[1, 0, 0, 2], [], [3, 0, 0, 4]]
```

`sympy.polys.densebasic.dmp_exclude(f, u, K)`
Exclude useless levels from f .
Return the levels excluded, the new excluded f , and the new u .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_exclude
```

```
>>> f = ZZ.map([[[1]], [[1], [2]]])
```

```
>>> dmp_exclude(f, 2, ZZ)
([2], [[1], [1, 2]], 1)
```

`sympy.polys.densebasic.dmp_include(f, J, u, K)`
Include useless levels in `f`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_include
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_include(f, [2], 1, ZZ)
[[[1]], [[1], [2]]]
```

`sympy.polys.densebasic.dmp_inject(f, u, K, front=False)`
Convert `f` from $K[X][Y]$ to $K[X, Y]$.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inject
```

```
>>> R, x, y = ring("x,y", ZZ)
```

```
>>> dmp_inject([R(1), x + 2], 0, R.to_domain())
([[[1]], [[1], [2]]], 2)
>>> dmp_inject([R(1), x + 2], 0, R.to_domain(), front=True)
([[[1]], [[1], [2]]], 2)
```

`sympy.polys.densebasic.dmp_eject(f, u, K, front=False)`
Convert `f` from $K[X, Y]$ to $K[X][Y]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_eject
```

```
>>> dmp_eject([[[1]], [[1], [2]]], 2, ZZ['x', 'y'])
[1, x + 2]
```

`sympy.polys.densebasic.dmp_terms_gcd(f, u, K)`
Remove GCD of terms from `f` in $K[X]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_terms_gcd
```

```
>>> f = ZZ.map([[1, 0], [1, 0, 0], [], []])
```

```
>>> dmp_terms_gcd(f, 1, ZZ)
((2, 1), [[1], [1, 0]])
```

`sympy.polys.densebasic.dmp_list_terms(f, u, K, order=None)`
List all non-zero terms from `f` in the given order `order`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_list_terms
```

```
>>> f = ZZ.map([[1, 1], [2, 3]])
```

```
>>> dmp_list_terms(f, 1, ZZ)
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
>>> dmp_list_terms(f, 1, ZZ, order='grevlex')
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
```

`sympy.polys.densebasic.dmp_apply_pairs(f, g, h, args, u, K)`
Apply `h` to pairs of coefficients of `f` and `g`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_apply_pairs
```

```
>>> h = lambda x, y, z: 2*x + y - z
```

```
>>> dmp_apply_pairs([[1], [2, 3]], [[3], [2, 1]], h, (1,), 1, ZZ)
[[4], [5, 6]]
```

`sympy.polys.densebasic.dmp_slice(f, m, n, u, K)`
Take a continuous subsequence of terms of `f` in $K[X]$.

`sympy.polys.densebasic.dup_random(n, a, b, K)`
Return a polynomial of degree `n` with coefficients in `[a, b]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_random
```

```
>>> dup_random(3, -10, 10, ZZ)
[-2, -8, 9, -4]
```

Arithmetic operations:

`sympy.polys.densearith.dmp_add_term(f, c, i, u, K)`
Add $c(x_2..x_u)*x_0^{**i}$ to f in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_term(x*y + 1, 2, 2)
2*x**2 + x*y + 1
```

`sympy.polys.densearith.dmp_sub_term(f, c, i, u, K)`
Subtract $c(x_2..x_u)*x_0^{**i}$ from f in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_term(2*x**2 + x*y + 1, 2, 2)
x*y + 1
```

`sympy.polys.densearith.dmp_mul_term(f, c, i, u, K)`
Multiply f by $c(x_2..x_u)*x_0^{**i}$ in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul_term(x**2*y + x, 3*y, 2)
3*x**4*y**2 + 3*x**3*y
```

`sympy.polys.densearith.dmp_add_ground(f, c, u, K)`
Add an element of the ground domain to f .

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x + 8
```


`sympy.polys.densearith.dmp_sub_ground(f, c, u, K)`
 Subtract an element of the ground domain from f .

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x
```

`sympy.polys.densearith.dmp_mul_ground(f, c, u, K)`
 Multiply f by a constant value in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul_ground(2*x + 2*y, ZZ(3))
6*x + 6*y
```

`sympy.polys.densearith.dmp_quo_ground(f, c, u, K)`
 Quotient by a constant in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, ZZ(2))
x**2*y + x
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, QQ(2))
x**2*y + 3/2*x
```

`sympy.polys.densearith.dmp_exquo_ground(f, c, u, K)`
 Exact quotient by a constant in $K[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_exquo_ground(x**2*y + 2*x, QQ(2))
1/2*x**2*y + x
```

`sympy.polys.densearith.dup_lshift(f, n, K)`
Efficiently multiply f by x^{**n} in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_lshift(x**2 + 1, 2)
x**4 + x**2
```

`sympy.polys.densearith.dup_rshift(f, n, K)`
Efficiently divide f by x^{**n} in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_rshift(x**4 + x**2, 2)
x**2 + 1
>>> R.dup_rshift(x**4 + x**2 + 2, 2)
x**2 + 1
```

`sympy.polys.densearith.dmp_abs(f, u, K)`
Make all coefficients positive in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_abs(x**2*y - x)
x**2*y + x
```

`sympy.polys.densearith.dmp_neg(f, u, K)`
Negate a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_neg(x**2*y - x)
-x**2*y + x
```

`sympy.polys.densearith.dmp_add(f, g, u, K)`
Add dense polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add(x**2 + y, x**2*y + x)
x**2*y + x**2 + x + y
```

`sympy.polys.densearith.dmp_sub(f, g, u, K)`
Subtract dense polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub(x**2 + y, x**2*y + x)
-x**2*y + x**2 - x + y
```

`sympy.polys.densearith.dmp_add_mul(f, g, h, u, K)`
Returns $f + g*h$ where f, g, h are in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_mul(x**2 + y, x, x + 2)
2*x**2 + 2*x + y
```

`sympy.polys.densearith.dmp_sub_mul(f, g, h, u, K)`
Returns $f - g*h$ where f, g, h are in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_mul(x**2 + y, x, x + 2)
-2*x + y
```

`sympy.polys.densearith.dmp_mul(f, g, u, K)`
Multiply dense polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul(x*y + 1, x)
x**2*y + x
```

`sympy.polys.densearith.dmp_sqr(f, u, K)`
Square dense polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sqr(x**2 + x*y + y**2)
x**4 + 2*x**3*y + 3*x**2*y**2 + 2*x*y**3 + y**4
```

`sympy.polys.densearith.dmp_pow(f, n, u, K)`
Raise f to the n -th power in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_pow(x*y + 1, 3)
x**3*y**3 + 3*x**2*y**2 + 3*x*y + 1
```

`sympy.polys.densearith.dmp_pdiv(f, g, u, K)`
Polynomial pseudo-division in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_pdiv(x**2 + x*y, 2*x + 2)
(2*x + 2*y - 2, -4*y + 4)
```

`sympy.polys.densearith.dmp_prem(f, g, u, K)`
Polynomial pseudo-remainder in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_prem(x**2 + x*y, 2*x + 2)
-4*y + 4
```

`sympy.polys.densearith.dmp_pquo(f, g, u, K)`
Polynomial exact pseudo-quotient in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2
```

```
>>> R.dmp_pquo(f, g)
2*x
```

```
>>> R.dmp_pquo(f, h)
2*x + 2*y - 2
```

`sympy.polys.densearith.dmp_pexquo(f, g, u, K)`
Polynomial pseudo-quotient in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2
```

```
>>> R.dmp_pexquo(f, g)
2*x
```

```
>>> R.dmp_pexquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

`sympy.polys.densearith.dmp_rr_div(f, g, u, K)`
Multivariate division with remainder over a ring.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_rr_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

`sympy.polys.densearith.dmp_ff_div(f, g, u, K)`
Polynomial division with remainder over a field.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_ff_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

`sympy.polys.densearith.dmp_div(f, g, u, K)`
Polynomial division with remainder in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

`sympy.polys.densearith.dmp_rem(f, g, u, K)`
Returns polynomial remainder in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
x**2 + x*y
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
-y + 1
```

`sympy.polys.densearith.dmp_quo(f, g, u, K)`
Returns exact polynomial quotient in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
0
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
1/2*x + 1/2*y - 1/2
```

`sympy.polys.densearith.dmp_exquo(f, g, u, K)`
Returns polynomial quotient in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x**2 + x*y
>>> g = x + y
>>> h = 2*x + 2
```

```
>>> R.dmp_exquo(f, g)
x
```

```
>>> R.dmp_exquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

`sympy.polys.densearith.dmp_max_norm(f, u, K)`
Returns maximum norm of a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_max_norm(2*x*y - x - 3)
3
```

`sympy.polys.densearith.dmp_l1_norm(f, u, K)`
Returns l1 norm of a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_l1_norm(2*x*y - x - 3)
6
```

`sympy.polys.densearith.dmp_expand(polys, u, K)`
Multiply together several polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_expand([x**2 + y**2, x + 1])
x**3 + x**2 + x*y**2 + y**2
```

Further tools:

`sympy.polys.denseutils.dmp_integrate(f, m, u, K)`
Computes the indefinite integral of f in x_0 in $K[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate(x + 2*y, 1)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate(x + 2*y, 2)
1/6*x**3 + x**2*y
```

`sympy.polys.denseutils.dmp_integrate_in(f, m, j, u, K)`
Computes the indefinite integral of f in x_j in $K[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate_in(x + 2*y, 1, 0)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate_in(x + 2*y, 1, 1)
x*y + y**2
```

`sympy.polys.denseutils.dmp_diff(f, m, u, K)`
 m -th order derivative in x_0 of a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff(f, 1)
y**2 + 2*y + 3
>>> R.dmp_diff(f, 2)
0
```


`sympy.polys.denseutils.dmp_diff_in(f, m, j, u, K)`
 m-th order derivative in x_j of a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff_in(f, 1, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_in(f, 1, 1)
2*x*y + 2*x + 4*y + 3
```

`sympy.polys.denseutils.dmp_eval(f, a, u, K)`
 Evaluate a polynomial at $x_0 = a$ in $K[X]$ using the Horner scheme.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_eval(2*x*y + 3*x + y + 2, 2)
5*y + 8
```

`sympy.polys.denseutils.dmp_eval_in(f, a, j, u, K)`
 Evaluate a polynomial at $x_j = a$ in $K[X]$ using the Horner scheme.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 2*x*y + 3*x + y + 2
```

```
>>> R.dmp_eval_in(f, 2, 0)
5*y + 8
>>> R.dmp_eval_in(f, 2, 1)
7*x + 4
```

`sympy.polys.denseutils.dmp_eval_tail(f, A, u, K)`
 Evaluate a polynomial at $x_j = a_j, \dots$ in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 2*x*y + 3*x + y + 2
```

```
>>> R.dmp_eval_tail(f, [2])
7*x + 4
>>> R.dmp_eval_tail(f, [2, 2])
18
```

`sympy.polys.denseutils.dmp_diff_eval_in(f, m, a, j, u, K)`
Differentiate and evaluate a polynomial in x_j at a in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff_eval_in(f, 1, 2, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_eval_in(f, 1, 2, 1)
6*x + 11
```

`sympy.polys.denseutils.dmp_trunc(f, p, u, K)`
Reduce a $K[X]$ polynomial modulo a polynomial p in $K[Y]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
>>> g = (y - 1).drop(x)
```

```
>>> R.dmp_trunc(f, g)
11*x**2 + 11*x + 5
```

`sympy.polys.denseutils.dmp_ground_trunc(f, p, u, K)`
Reduce a $K[X]$ polynomial modulo a constant p in K .

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
```

```
>>> R.dmp_ground_trunc(f, ZZ(3))
-x**2 - x*y - y
```

`sympy.polys.denseutils.dup_monic(f, K)`
 Divide all coefficients by $\text{LC}(f)$ in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> R.dup_monic(3*x**2 + 6*x + 9)
x**2 + 2*x + 3
```

```
>>> R, x = ring("x", QQ)
>>> R.dup_monic(3*x**2 + 4*x + 2)
x**2 + 4/3*x + 2/3
```

`sympy.polys.denseutils.dmp_ground_monic(f, u, K)`
 Divide all coefficients by $\text{LC}(f)$ in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 3*x**2*y + 6*x**2 + 3*x*y + 9*y + 3
```

```
>>> R.dmp_ground_monic(f)
x**2*y + 2*x**2 + x*y + 3*y + 1
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
```

```
>>> R.dmp_ground_monic(f)
x**2*y + 8/3*x**2 + 5/3*x*y + 2*x + 2/3*y + 1
```

`sympy.polys.denseutils.dup_content(f, K)`
 Compute the GCD of coefficients of f in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_content(f)
2
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_content(f)
2
```

`sympy.polys.densetools.dmp_ground_content(f, u, K)`
Compute the GCD of coefficients of f in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

`sympy.polys.densetools.dup_primitive(f, K)`
Compute content and the primitive form of f in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

`sympy.polys.densetools.dmp_ground_primitive(f, u, K)`
Compute content and the primitive form of f in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

`sympy.polys.denseutils.dup_extract(f, g, K)`
Extract common content from a pair of polynomials in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_extract(6*x**2 + 12*x + 18, 4*x**2 + 8*x + 12)
(2, 3*x**2 + 6*x + 9, 2*x**2 + 4*x + 6)
```

`sympy.polys.denseutils.dmp_ground_extract(f, g, u, K)`
Extract common content from a pair of polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_ground_extract(6*x*y + 12*x + 18, 4*x*y + 8*x + 12)
(2, 3*x*y + 6*x + 9, 2*x*y + 4*x + 6)
```

`sympy.polys.denseutils.dup_real_imag(f, K)`
Return bivariate polynomials f_1 and f_2 , such that $f = f_1 + f_2*I$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dup_real_imag(x**3 + x**2 + x + 1)
(x**3 + x**2 - 3*x*y**2 + x - y**2 + 1, 3*x**2*y + 2*x*y - y**3 + y)
```

`sympy.polys.denseutils.dup_mirror(f, K)`
Evaluate efficiently the composition $f(-x)$ in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_mirror(x**3 + 2*x**2 - 4*x + 2)
-x**3 + 2*x**2 + 4*x + 2
```

`sympy.polys.densetools.dup_scale(f, a, K)`
Evaluate efficiently composition $f(a*x)$ in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_scale(x**2 - 2*x + 1, ZZ(2))
4*x**2 - 4*x + 1
```

`sympy.polys.densetools.dup_shift(f, a, K)`
Evaluate efficiently Taylor shift $f(x + a)$ in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_shift(x**2 - 2*x + 1, ZZ(2))
x**2 + 2*x + 1
```

`sympy.polys.densetools.dup_transform(f, p, q, K)`
Evaluate functional transformation $q**n * f(p/q)$ in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_transform(x**2 - 2*x + 1, x**2 + 1, x - 1)
x**4 - 2*x**3 + 5*x**2 - 4*x + 4
```

`sympy.polys.densetools.dmp_compose(f, g, u, K)`
Evaluate functional composition $f(g)$ in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_compose(x*y + 2*x + y, y)
y**2 + 3*y
```

`sympy.polys.denseutils.dup_decompose(f, K)`

Computes functional decomposition of f in $K[x]$.

Given a univariate polynomial f with coefficients in a field of characteristic zero, returns list $[f_1, f_2, \dots, f_n]$, where:

```
f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n))
```

and $f_2, \dots, f_n$ are monic and homogeneous polynomials of at least second degree.

Unlike factorization, complete functional decompositions of polynomials are not unique, consider examples:

1. $f \circ g = f(x + b) \circ (g - b)$
2. $x^{**n} \circ x^{**m} = x^{**m} \circ x^{**n}$
3. $T_n \circ T_m = T_m \circ T_n$

where T_n and T_m are Chebyshev polynomials.

References

1. [Kozen89] (page 1866)

[Kozen89] (page 1866)

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_decompose(x**4 - 2*x**3 + x**2)
[x**2, x**2 - x]
```

`sympy.polys.denseutils.dmp_lift(f, u, K)`

Convert algebraic coefficients to integers in $K[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> from sympy import I
```

```
>>> K = QQ.algebraic_field(I)
>>> R, x = ring("x", K)
```

```
>>> f = x**2 + K([QQ(1), QQ(0)])*x + K([QQ(2), QQ(0)])
```

```
>>> R.dmp_lift(f)
x**8 + 2*x**6 + 9*x**4 - 8*x**2 + 16
```

`sympy.polys.densetools.dup_sign_variations(f, K)`
Compute the number of sign variations of f in $K[x]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_sign_variations(x**4 - x**2 - x + 1)
2
```

`sympy.polys.densetools.dmp_clear_denoms(f, u, K0, K1=None, convert=False)`
Clear denominators, i.e. transform K_0 to K_1 .

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + QQ(1,3)*y + 1
```

```
>>> R.dmp_clear_denoms(f, convert=False)
(6, 3*x + 2*y + 6)
>>> R.dmp_clear_denoms(f, convert=True)
(6, 3*x + 2*y + 6)
```

`sympy.polys.densetools.dmp_revert(f, g, u, K)`
Compute $f^{**}(-1) \bmod x^{**}n$ using Newton iteration.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

Manipulation of dense, univariate polynomials with finite field coefficients

Functions in this module carry the prefix `gf_`, referring to the classical name “Galois Fields” for finite fields. Note that many polynomial factorization algorithms work by reduction to the finite field case, so having special implementations for this case is justified both by performance, and by the necessity of certain methods which do not even make sense over general fields.

`sympy.polys.galoistools.gf_crt(U, M, K=None)`
Chinese Remainder Theorem.

Given a set of integer residues $u_0, \dots, u_n$ and a set of co-prime integer moduli $m_0, \dots, m_n$, returns an integer u , such that $u = u_i \bmod m_i$ for $i = 0, \dots, n$.

As an example consider a set of residues $U = [49, 76, 65]$ and a set of moduli $M = [99, 97, 95]$. Then we have:


```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt
>>> from sympy.ntheory.modular import solve_congruence

>>> gf_crt([49, 76, 65], [99, 97, 95], ZZ)
639985
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

Note: this is a low-level routine with no error checking.

See also:

[`sympy.ntheory.modular.crt`](#) (page 322) a higher level crt routine

[`sympy.ntheory.modular.solve\_congruence`](#) (page 323)

`sympy.polys.galoistools.gf_crt1(M, K)`

First part of the Chinese Remainder Theorem.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt1
```

```
>>> gf_crt1([99, 97, 95], ZZ)
(912285, [9215, 9405, 9603], [62, 24, 12])
```

`sympy.polys.galoistools.gf_crt2(U, M, p, E, S, K)`

Second part of the Chinese Remainder Theorem.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt2
```

```
>>> U = [49, 76, 65]
>>> M = [99, 97, 95]
>>> p = 912285
>>> E = [9215, 9405, 9603]
>>> S = [62, 24, 12]
```

```
>>> gf_crt2(U, M, p, E, S, ZZ)
639985
```

`sympy.polys.galoistools.gf_int(a, p)`

Coerce $a \bmod p$ to an integer in the range $[-p/2, p/2]$.

Examples

```
>>> from sympy.polys.galoistools import gf_int
```

```
>>> gf_int(2, 7)
2
>>> gf_int(5, 7)
-2
```

`sympy.polys.galoistools.gf_degree(f)`
Return the leading degree of `f`.

Examples

```
>>> from sympy.polys.galoistools import gf_degree
```

```
>>> gf_degree([1, 1, 2, 0])
3
>>> gf_degree([])
-1
```

`sympy.polys.galoistools.gf_LC(f, K)`
Return the leading coefficient of `f`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_LC
```

```
>>> gf_LC([3, 0, 1], ZZ)
3
```

`sympy.polys.galoistools.gf_TC(f, K)`
Return the trailing coefficient of `f`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_TC
```

```
>>> gf_TC([3, 0, 1], ZZ)
1
```

`sympy.polys.galoistools.gf_strip(f)`
Remove leading zeros from `f`.

Examples

```
>>> from sympy.polys.galoistools import gf_strip
```

```
>>> gf_strip([0, 0, 0, 3, 0, 1])
[3, 0, 1]
```

`sympy.polys.galoistools.gf_trunc(f, p)`
Reduce all coefficients modulo p .

Examples

```
>>> from sympy.polys.galoistools import gf_trunc
```

```
>>> gf_trunc([7, -2, 3], 5)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_normal(f, p, K)`
Normalize all coefficients in K .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_normal
```

```
>>> gf_normal([5, 10, 21, -3], 5, ZZ)
[1, 2]
```

`sympy.polys.galoistools.gf_from_dict(f, p, K)`
Create a $\text{GF}(p)[x]$ polynomial from a dict.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_dict
```

```
>>> gf_from_dict({10: ZZ(4), 4: ZZ(33), 0: ZZ(-1)}, 5, ZZ)
[4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4]
```

`sympy.polys.galoistools.gf_to_dict(f, p, symmetric=True)`
Convert a $\text{GF}(p)[x]$ polynomial to a dict.

Examples

```
>>> from sympy.polys.galoistools import gf_to_dict
```

```
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5)
{0: -1, 4: -2, 10: -1}
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5, symmetric=False)
{0: 4, 4: 3, 10: 4}
```

`sympy.polys.galoistools.gf_from_int_poly(f, p)`

Create a $\text{GF}(p)[x]$ polynomial from $\mathbb{Z}[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_int_poly
```

```
>>> gf_from_int_poly([7, -2, 3], 5)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_to_int_poly(f, p, symmetric=True)`

Convert a $\text{GF}(p)[x]$ polynomial to $\mathbb{Z}[x]$.

Examples

```
>>> from sympy.polys.galoistools import gf_to_int_poly
```

```
>>> gf_to_int_poly([2, 3, 3], 5)
[2, -2, -2]
>>> gf_to_int_poly([2, 3, 3], 5, symmetric=False)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_neg(f, p, K)`

Negate a polynomial in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_neg
```

```
>>> gf_neg([3, 2, 1, 0], 5, ZZ)
[2, 3, 4, 0]
```

`sympy.polys.galoistools.gf_add_ground(f, a, p, K)`

Compute $f + a$ where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_ground
```

```
>>> gf_add_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 1]
```

`sympy.polys.galoistools.gf_sub_ground(f, a, p, K)`
 Compute $f - a$ where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_ground
```

```
>>> gf_sub_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 2]
```

`sympy.polys.galoistools.gf_mul_ground(f, a, p, K)`
 Compute $f * a$ where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul_ground
```

```
>>> gf_mul_ground([3, 2, 4], 2, 5, ZZ)
[1, 4, 3]
```

`sympy.polys.galoistools.gf_quo_ground(f, a, p, K)`
 Compute f/a where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo_ground
```

```
>>> gf_quo_ground(ZZ.map([3, 2, 4]), ZZ(2), 5, ZZ)
[4, 1, 2]
```

`sympy.polys.galoistools.gf_add(f, g, p, K)`
 Add polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add
```

```
>>> gf_add([3, 2, 4], [2, 2, 2], 5, ZZ)
[4, 1]
```

`sympy.polys.galoistools.gf_sub(f, g, p, K)`
 Subtract polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub
```

```
>>> gf_sub([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 2]
```

`sympy.polys.galoistools.gf_mul(f, g, p, K)`
Multiply polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul
```

```
>>> gf_mul([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 3, 2, 3]
```

`sympy.polys.galoistools.gf_sqr(f, p, K)`
Square polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqr
```

```
>>> gf_sqr([3, 2, 4], 5, ZZ)
[4, 2, 3, 1, 1]
```

`sympy.polys.galoistools.gf_add_mul(f, g, h, p, K)`
Returns $f + g \cdot h$ where f, g, h in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_mul
>>> gf_add_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[2, 3, 2, 2]
```

`sympy.polys.galoistools.gf_sub_mul(f, g, h, p, K)`
Compute $f - g \cdot h$ where f, g, h in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_mul
```

```
>>> gf_sub_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[3, 3, 2, 1]
```

`sympy.polys.galoistools.gf_expand(F, p, K)`
Expand results of `factor()` in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_expand
```

```
>>> gf_expand([(3, 2, 4), 1], ([2, 2], 2), ([3, 1], 3), 5, ZZ)
[4, 3, 0, 3, 0, 1, 4, 1]
```

`sympy.polys.galoistools.gf_div(f, g, p, K)`
Division with remainder in $\text{GF}(p)[x]$.

Given univariate polynomials f and g with coefficients in a finite field with p elements, returns polynomials q and r (quotient and remainder) such that $f = qg + r$.

Consider polynomials $x^3 + x + 1$ and $x^2 + x$ in $\text{GF}(2)$:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_div, gf_add_mul

>>> gf_div(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
([1, 1], [1])
```

As result we obtained quotient $x + 1$ and remainder 1, thus:

```
>>> gf_add_mul(ZZ.map([1]), ZZ.map([1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 0, 1, 1]
```

References

1. [Monagan93] (page 1866)
2. [Gathen99] (page 1866)

[Monagan93] (page 1866), [Gathen99] (page 1866)

`sympy.polys.galoistools.gf_rem(f, g, p, K)`
Compute polynomial remainder in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rem
```

```
>>> gf_rem(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1]
```

`sympy.polys.galoistools.gf_quo(f, g, p, K)`
Compute exact quotient in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo
```

```
>>> gf_quo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 1]
>>> gf_quo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

`sympy.polys.galoistools.gf_exquo(f, g, p, K)`
Compute polynomial quotient in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_exquo
```

```
>>> gf_exquo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

```
>>> gf_exquo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
Traceback (most recent call last):
...
ExactQuotientFailed: [1, 1, 0] does not divide [1, 0, 1, 1]
```

`sympy.polys.galoistools.gf_lshift(f, n, K)`
Efficiently multiply f by x^{*n} .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lshift
```

```
>>> gf_lshift([3, 2, 4], 4, ZZ)
[3, 2, 4, 0, 0, 0, 0]
```

`sympy.polys.galoistools.gf_rshift(f, n, K)`
Efficiently divide f by x^{*n} .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rshift
```

```
>>> gf_rshift([1, 2, 3, 4, 0], 3, ZZ)
([1, 2], [3, 4, 0])
```

`sympy.polys.galoistools.gf_pow(f, n, p, K)`
Compute f^{*n} in $\text{GF}(p)[x]$ using repeated squaring.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow
```

```
>>> gf_pow([3, 2, 4], 3, 5, ZZ)
[2, 4, 4, 2, 2, 1, 4]
```

`sympy.polys.galoistools.gf_pow_mod(f, n, g, p, K)`

Compute f^n in $\text{GF}(p)[x]/(g)$ using repeated squaring.

Given polynomials f and g in $\text{GF}(p)[x]$ and a non-negative integer n , efficiently computes $f^n \pmod{g}$ i.e. the remainder of f^n from division by g , using the repeated squaring algorithm.

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow_mod
```

```
>>> gf_pow_mod(ZZ.map([3, 2, 4]), 3, ZZ.map([1, 1]), 5, ZZ)
[]
```

`sympy.polys.galoistools.gf_gcd(f, g, p, K)`

Euclidean Algorithm in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcd
```

```
>>> gf_gcd(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 3]
```

`sympy.polys.galoistools.gf_lcm(f, g, p, K)`

Compute polynomial LCM in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lcm
```

```
>>> gf_lcm(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 2, 0, 4]
```

`sympy.polys.galoistools.gf_cofactors(f, g, p, K)`
Compute polynomial GCD and cofactors in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_cofactors
```

```
>>> gf_cofactors(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
([1, 3], [3, 3], [2, 1])
```

`sympy.polys.galoistools.gf_gcdex(f, g, p, K)`
Extended Euclidean Algorithm in $\text{GF}(p)[x]$.

Given polynomials f and g in $\text{GF}(p)[x]$, computes polynomials s , t and h , such that $h = \text{gcd}(f, g)$ and $s*f + t*g = h$. The typical application of EEA is solving polynomial diophantine equations.

Consider polynomials $f = (x + 7)(x + 1)$, $g = (x + 7)(x^2 + 1)$ in $\text{GF}(11)[x]$. Application of Extended Euclidean Algorithm gives:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcdex, gf_mul, gf_add

>>> s, t, g = gf_gcdex(ZZ.map([1, 8, 7]), ZZ.map([1, 7, 1, 7]), 11, ZZ)
>>> s, t, g
([5, 6], [6], [1, 7])
```

As result we obtained polynomials $s = 5x + 6$ and $t = 6$, and additionally $\text{gcd}(f, g) = x + 7$. This is correct because:

```
>>> S = gf_mul(s, ZZ.map([1, 8, 7]), 11, ZZ)
>>> T = gf_mul(t, ZZ.map([1, 7, 1, 7]), 11, ZZ)

>>> gf_add(S, T, 11, ZZ) == [1, 7]
True
```

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

`sympy.polys.galoistools.gf_monic(f, p, K)`
Compute LC and a monic polynomial in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_monic
```

```
>>> gf_monic(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [1, 4, 3])
```

`sympy.polys.galoistools.gf_diff(f, p, K)`
Differentiate polynomial in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_diff
```

```
>>> gf_diff([3, 2, 4], 5, ZZ)
[1, 2]
```

`sympy.polys.galoistools.gf_eval(f, a, p, K)`
Evaluate $f(a)$ in $\text{GF}(p)$ using Horner scheme.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_eval
```

```
>>> gf_eval([3, 2, 4], 2, 5, ZZ)
0
```

`sympy.polys.galoistools.gf_multi_eval(f, A, p, K)`
Evaluate $f(a)$ for a in $[a_1, \dots, a_n]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_multi_eval
```

```
>>> gf_multi_eval([3, 2, 4], [0, 1, 2, 3, 4], 5, ZZ)
[4, 4, 0, 2, 0]
```

`sympy.polys.galoistools.gf_compose(f, g, p, K)`
Compute polynomial composition $f(g)$ in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose
```

```
>>> gf_compose([3, 2, 4], [2, 2, 2], 5, ZZ)
[2, 4, 0, 3, 0]
```

`sympy.polys.galoistools.gf_compose_mod(g, h, f, p, K)`
Compute polynomial composition $g(h)$ in $\text{GF}(p)[x]/(f)$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose_mod
```

```
>>> gf_compose_mod(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 2]), ZZ.map([4, 3]), 5, ZZ)
[4]
```

`sympy.polys.galoistools.gf_trace_map(a, b, c, n, f, p, K)`
Compute polynomial trace map in $\text{GF}(p)[x]/(f)$.

Given a polynomial f in $\text{GF}(p)[x]$, polynomials a, b, c in the quotient ring $\text{GF}(p)[x]/(f)$ such that $b = c^{**t} \pmod{f}$ for some positive power t of p , and a positive integer n , returns a mapping:

```
a -> a**t**n, a + a**t + a**t**2 + ... + a**t**n (mod f)
```

In factorization context, $b = x^{**p} \pmod{f}$ and $c = x \pmod{f}$. This way we can efficiently compute trace polynomials in equal degree factorization routine, much faster than with other methods, like iterated Frobenius algorithm, for large degrees.

References

1. [Gathen92] (page 1866)

[Gathen92] (page 1866)

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_trace_map
```

```
>>> gf_trace_map([1, 2], [4, 4], [1, 1], 4, [3, 2, 4], 5, ZZ)
([1, 3], [1, 3])
```

`sympy.polys.galoistools.gf_random(n, p, K)`
Generate a random polynomial in $\text{GF}(p)[x]$ of degree n .

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_random
>>> gf_random(10, 5, ZZ)
[1, 2, 3, 2, 1, 1, 1, 2, 0, 4, 2]
```

`sympy.polys.galoistools.gf_irreducible(n, p, K)`
 Generate random irreducible polynomial of degree n in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible
>>> gf_irreducible(10, 5, ZZ)
[1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]
```

`sympy.polys.galoistools.gf_irreducible_p(f, p, K)`
 Test irreducibility of a polynomial f in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible_p
```

```
>>> gf_irreducible_p(ZZ.map([1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]), 5, ZZ)
True
>>> gf_irreducible_p(ZZ.map([3, 2, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_p(f, p, K)`
 Return True if f is square-free in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_p
```

```
>>> gf_sqf_p(ZZ.map([3, 2, 4]), 5, ZZ)
True
>>> gf_sqf_p(ZZ.map([2, 4, 4, 2, 2, 1, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_part(f, p, K)`
 Return square-free part of a $\text{GF}(p)[x]$ polynomial.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_part
```

```
>>> gf_sqf_part(ZZ.map([1, 1, 3, 0, 1, 0, 2, 2, 1]), 5, ZZ)
[1, 4, 3]
```

`sympy.polys.galoistools.gf_sqf_list(f, p, K, all=False)`
 Return the square-free decomposition of a $\text{GF}(p)[x]$ polynomial.

Given a polynomial f in $\text{GF}(p)[x]$, returns the leading coefficient of f and a square-free decomposition $f_1^{e_1} f_2^{e_2} \dots f_k^{e_k}$ such that all f_i are monic polynomials and (f_i, f_j) for $i \neq j$ are co-prime and $e_1 \dots e_k$ are given in increasing order. All trivial terms (i.e. $f_i = 1$) aren't included in the output.

Consider polynomial $f = x^{11} + 1$ over $\text{GF}(11)[x]$:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import (
...     gf_from_dict, gf_diff, gf_sqf_list, gf_pow,
... )
...
>>> f = gf_from_dict({11: ZZ(1), 0: ZZ(1)}, 11, ZZ)
```

Note that $f'(x) = 0$:

```
>>> gf_diff(f, 11, ZZ)
[]
```

This phenomenon doesn't happen in characteristic zero. However we can still compute square-free decomposition of f using `gf_sqf()`:

```
>>> gf_sqf_list(f, 11, ZZ)
(1, [[1, 1], 11])
```

We obtained factorization $f = (x + 1)^{11}$. This is correct because:

```
>>> gf_pow([1, 1], 11, 11, ZZ) == f
True
```

References

1. [Geddes92] (page 1866)

[Geddes92] (page 1866)

`sympy.polys.galoistools.gf_Qmatrix(f, p, K)`
Calculate Berlekamp's Q matrix.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix
```

```
>>> gf_Qmatrix([3, 2, 4], 5, ZZ)
[[1, 0],
 [3, 4]]
```

```
>>> gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 0, 0],
 [0, 4, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 4]]
```

`sympy.polys.galoistools.gf_Qbasis(Q, p, K)`
 Compute a basis of the kernel of Q.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix, gf_Qbasis
```

```
>>> gf_Qbasis(gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ), 5, ZZ)
[[1, 0, 0, 0], [0, 0, 1, 0]]
```

```
>>> gf_Qbasis(gf_Qmatrix([3, 2, 4], 5, ZZ), 5, ZZ)
[[1, 0]]
```

`sympy.polys.galoistools.gf_berlekamp(f, p, K)`
 Factor a square-free f in $\text{GF}(p)[x]$ for small p.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_berlekamp
```

```
>>> gf_berlekamp([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 2], [1, 0, 3]]
```

`sympy.polys.galoistools.gf_zassenhaus(f, p, K)`
 Factor a square-free f in $\text{GF}(p)[x]$ for medium p.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_zassenhaus
```

```
>>> gf_zassenhaus(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

`sympy.polys.galoistools.gf_shoup(f, p, K)`
 Factor a square-free f in $\text{GF}(p)[x]$ for large p.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_shoup
```

```
>>> gf_shoup(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

`sympy.polys.galoistools.gf_factor_sqf(f, p, K, method=None)`
 Factor a square-free polynomial f in $\text{GF}(p)[x]$.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor_sqf
```

```
>>> gf_factor_sqf(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [[1, 1], [1, 3]])
```

`sympy.polys.galoistools.gf_factor(f, p, K)`
Factor (non square-free) polynomials in $\text{GF}(p)[x]$.

Given a possibly non square-free polynomial f in $\text{GF}(p)[x]$, returns its complete factorization into irreducibles:

```
f_1(x)**e_1 f_2(x)**e_2 ... f_d(x)**e_d
```

where each f_i is a monic polynomial and $\text{gcd}(f_i, f_j) == 1$, for $i \neq j$. The result is given as a tuple consisting of the leading coefficient of f and a list of factors of f with their multiplicities.

The algorithm proceeds by first computing square-free decomposition of f and then iteratively factoring each of square-free factors.

Consider a non square-free polynomial $f = (7x + 1)(x + 2)^2$ in $\text{GF}(11)[x]$. We obtain its factorization into irreducibles as follows:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor

>>> gf_factor(ZZ.map([5, 2, 7, 2]), 11, ZZ)
(5, [[1, 2], 1], [[1, 8], 2])
```

We arrived with factorization $f = 5(x + 2)(x + 8)^2$. We didn't recover the exact form of the input polynomial because we requested to get monic factors of f and its leading coefficient separately.

Square-free factors of f can be factored into irreducibles over $\text{GF}(p)$ using three very different methods:

Berlekamp efficient for very small values of p (usually $p < 25$)

Cantor-Zassenhaus efficient on average input and with "typical" p

Shoup-Kaltofen-Gathen efficient with very large inputs and modulus

If you want to use a specific factorization method, instead of the default one, set `GF_FACTOR_METHOD` with one of `berlekamp`, `zassenhaus` or `shoup` values.

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

`sympy.polys.galoistools.gf_value(f, a)`
Value of polynomial 'f' at 'a' in field R.

Examples

```
>>> from sympy.polys.galoistools import gf_value
```

```
>>> gf_value([1, 7, 2, 4], 11)
2204
```

`sympy.polys.galoistools.gf_csolve(f, n)`

To solve $f(x) \equiv 0 \pmod{n}$.

n is divided into canonical factors and $f(x) \equiv 0 \pmod{p^e}$ will be solved for each factor. Applying the Chinese Remainder Theorem to the results returns the final answers.

References

[1] ‘An introduction to the Theory of Numbers’ 5th Edition by Ivan Niven, Zuckerman and Montgomery.

Examples

Solve $[1, 1, 7] \pmod{189}$:

```
>>> from sympy.polys.galoistools import gf_csolve
>>> gf_csolve([1, 1, 7], 189)
[13, 49, 76, 112, 139, 175]
```

Manipulation of sparse, distributed polynomials and vectors

Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a sparse representation.

Sparse polynomials are represented as dictionaries.

`sympy.polys.rings.Ring(symbols, domain, order=LexOrder())`

Construct a polynomial ring returning $(\text{ring}, x_1, \dots, x_n)$.

Parameters **symbols** : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

domain : Domain or coercible

order : Order or coercible, optional, defaults to lex

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> R, x, y, z = ring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.xring`(symbols, domain, order=LexOrder())
Construct a polynomial ring returning (ring, (x₁, ..., x_n)).

Parameters **symbols** : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

domain : Domain or coercible

order : Order or coercible, optional, defaults to lex

Examples

```
>>> from sympy.polys.rings import xring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> R, (x, y, z) = xring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.vring`(symbols, domain, order=LexOrder())
Construct a polynomial ring and inject x₁, ..., x_n into the global namespace.

Parameters **symbols** : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

domain : Domain or coercible

order : Order or coercible, optional, defaults to lex

Examples

```
>>> from sympy.polys.rings import vring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> vring("x,y,z", ZZ, lex)
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.sring`(exprs, *symbols, **options)

Construct a ring deriving generators and domain from options and input expressions.

Parameters **exprs** : Expr or sequence of Expr (sympifiable)

symbols : sequence of Symbol/Expr

options : keyword arguments understood by Options

Examples

```
>>> from sympy.core import symbols
>>> from sympy.polys.rings import sring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> x, y, z = symbols("x,y,z")
>>> R, f = sring(x + 2*y + 3*z)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> f
x + 2*y + 3*z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

class `sympy.polys.rings.PolyRing`

Multivariate distributed polynomial ring.

Attributes

domain	
gens	
ngens	
order	
symbols	

add(*objs)

Add a sequence of polynomials or containers of polynomials.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
>>> R.add([ x**2 + 2*i + 3 for i in range(4) ])
4*x**2 + 24
>>> _.factor_list()
(4, [(x**2 + 6, 1)])
```

add_gens(symbols)

Add the elements of symbols as generators to self

compose(other)
Add the generators of other to self

drop(*gens)
Remove specified generators from this ring.

drop_to_ground(*gens)
Remove specified generators from the ring and inject them into its domain.

index(gen)
Compute index of gen in self.gens.

monomial_basis(i)
Return the ith-basis element.

mul(*objs)
Multiply a sequence of polynomials or containers of polynomials.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
>>> R.mul([ x**2 + 2*i + 3 for i in range(4) ])
x**8 + 24*x**6 + 206*x**4 + 744*x**2 + 945
>>> _factor_list()
(1, [(x**2 + 3, 1), (x**2 + 5, 1), (x**2 + 7, 1), (x**2 + 9, 1)])
```

class sympy.polys.rings.PolyElement
Element of multivariate distributed polynomial ring.

almosteq(p2, tolerance=None)
Approximate equality test for polynomials.

cancel(g)
Cancel common factors in a rational function f/g.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> (2*x**2 - 2).cancel(x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

coeff(element)
Returns the coefficient that stands next to the given monomial.

Parameters **element** : PolyElement (with `is_monomial = True`) or 1

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y, z = ring("x,y,z", ZZ)
>>> f = 3*x**2*y - x*y*z + 7*z**3 + 23
```

```
>>> f.coeff(x**2*y)
3
>>> f.coeff(x*y)
0
>>> f.coeff(1)
23
```

coeffs(order=None)

Ordered list of polynomial coefficients.

Parameters **order** : Order or coercible, optional

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.coeffs()
[2, 1]
>>> f.coeffs(grlex)
[1, 2]
```

const()

Returns the constant coefficient.

content()

Returns GCD of polynomial's coefficients.

copy()

Return a copy of polynomial self.

Polynomials are mutable; if one is interested in preserving a polynomial, and one plans to use inplace operations, one can copy the polynomial. This method makes a shallow copy.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.rings import ring
```

```
>>> R, x, y = ring('x, y', ZZ)
>>> p = (x + y)**2
>>> p1 = p.copy()
>>> p2 = p
>>> p[R.zero_monom] = 3
>>> p
```

(continues on next page)

(continued from previous page)

```

x**2 + 2*x*y + y**2 + 3
>>> p1
x**2 + 2*x*y + y**2
>>> p2
x**2 + 2*x*y + y**2 + 3

```

degree(x=None)

The leading degree in x or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object `-oo`).

degrees()

A tuple containing leading degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object `-oo`)

diff(x)

Computes partial derivative in x .

Examples

```

>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

```

```

>>> _, x, y = ring("x,y", ZZ)
>>> p = x + x**2*y**3
>>> p.diff(x)
2*x*y**3 + 1

```

div(fv)

Division algorithm, see [CLO] p64.

fv array of polynomials return qv, r such that $\text{self} = \sum(\text{fv}[i]*qv[i]) + r$

All polynomials are required not to be Laurent polynomials.

Examples

```

>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

```

```

>>> _, x, y = ring('x, y', ZZ)
>>> f = x**3
>>> f0 = x - y**2
>>> f1 = x - y
>>> qv, r = f.div((f0, f1))
>>> qv[0]
x**2 + x*y**2 + y**4
>>> qv[1]
0
>>> r
y**6

```

imul_num(c)

multiply inplace the polynomial p by an element in the coefficient ring, provided p is not one of the generators; else multiply not inplace

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p1 = p.imul_num(3)
>>> p1
3*x + 3*y**2
>>> p1 is p
True
>>> p = x
>>> p1 = p.imul_num(3)
>>> p1
3*x
>>> p1 is p
False
```

itercoeffs()

Iterator over coefficients of a polynomial.

itermonoms()

Iterator over monomials of a polynomial.

iterterms()

Iterator over terms of a polynomial.

leading_expv()

Leading monomial tuple according to the monomial ordering.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y, z = ring('x, y, z', ZZ)
>>> p = x**4 + x**3*y + x**2*z**2 + z**7
>>> p.leading_expv()
(4, 0, 0)
```

leading_monom()

Leading monomial as a polynomial element.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_monom()
x*y
```

leading_term()

Leading term as a polynomial element.

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_term()
3*x*y
```

listcoeffs()

Unordered list of polynomial coefficients.

listmonoms()

Unordered list of polynomial monomials.

listterms()

Unordered list of polynomial terms.

monic()

Divides all coefficients by the leading coefficient.

monoms(order=None)

Ordered list of polynomial monomials.

Parameters **order** : Order or coercible, optional

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.monoms()
[(2, 3), (1, 7)]
>>> f.monoms(grlex)
[(1, 7), (2, 3)]
```

primitive()

Returns content and a primitive polynomial.

square()

square of a polynomial

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p.square()
x**2 + 2*x*y**2 + y**4
```

strip_zero()

Eliminate monomials with zero coefficient.

tail_degree(x=None)

The tail degree in x or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

tail_degrees()

A tuple containing tail degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

terms(order=None)

Ordered list of polynomial terms.

Parameters **order** : Order or coercible, optional

Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.terms()
[((2, 3), 2), ((1, 7), 1)]
>>> f.terms(grlex)
[((1, 7), 1), ((2, 3), 2)]
```

In commutative algebra, one often studies not only polynomials, but also modules over polynomial rings. The polynomial manipulation module provides rudimentary low-level support for finitely generated free modules. This is mainly used for Groebner basis computations (see there), so manipulation functions are only provided to the extend needed. They carry the prefix `sdm_`. Note that in examples, the generators of the free module are called $f_1, f_2, \dots$

`sympy.polys.distributedmodules.sdm_monomial_mul(M, X)`

Multiply tuple X representing a monomial of $K[X]$ into the tuple M representing a monomial of F .

Examples

Multiplying xy^3 into xf_1 yields $x^2y^3f_1$:

```
>>> from sympy.polys.distributedmodules import sdm_monomial_mul
>>> sdm_monomial_mul((1, 1, 0), (1, 3))
(1, 2, 3)
```

`sympy.polys.distributedmodules.sdm_monomial_deg(M)`
Return the total degree of M.

Examples

For example, the total degree of x^2yf_5 is 3:

```
>>> from sympy.polys.distributedmodules import sdm_monomial_deg
>>> sdm_monomial_deg((5, 2, 1))
3
```

`sympy.polys.distributedmodules.sdm_monomial_divides(A, B)`
Does there exist a (polynomial) monomial X such that $XA = B$?

Examples

Positive examples:

In the following examples, the monomial is given in terms of x, y and the generator(s), `f_1`, `f_2` etc. The tuple form of that monomial is used in the call to `sdm_monomial_divides`. Note: the generator appears last in the expression but first in the tuple and other factors appear in the same order that they appear in the monomial expression.

$A = f_1$ divides $B = f_1$

```
>>> from sympy.polys.distributedmodules import sdm_monomial_divides
>>> sdm_monomial_divides((1, 0, 0), (1, 0, 0))
True
```

$A = f_1$ divides $B = x^2yf_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

$A = xyf_5$ divides $B = x^2yf_5$

```
>>> sdm_monomial_divides((5, 1, 1), (5, 2, 1))
True
```

Negative examples:

$A = f_1$ does not divide $B = f_2$

```
>>> sdm_monomial_divides((1, 0, 0), (2, 0, 0))
False
```

$A = xf_1$ does not divide $B = f_1$

```
>>> sdm_monomial_divides((1, 1, 0), (1, 0, 0))
False
```

$A = xy^2f_5$ does not divide $B = yf_5$

```
>>> sdm_monomial_divides((5, 1, 2), (5, 0, 1))
False
```

`sympy.polys.distributedmodules.sdm_LC(f, K)`

Returns the leading coefficient of f .

`sympy.polys.distributedmodules.sdm_to_dict(f)`

Make a dictionary from a distributed polynomial.

`sympy.polys.distributedmodules.sdm_from_dict(d, O)`

Create an sdm from a dictionary.

Here O is the monomial order to use.

```
>>> from sympy.polys.distributedmodules import sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 1, 0): QQ(1), (1, 0, 0): QQ(2), (0, 1, 0): QQ(0)}
>>> sdm_from_dict(dic, lex)
[((1, 1, 0), 1), ((1, 0, 0), 2)]
```

`sympy.polys.distributedmodules.sdm_add(f, g, O, K)`

Add two module elements f, g .

Addition is done over the ground field K , monomials are ordered according to O .

Examples

All examples use lexicographic order.

$$(xyf_1) + (f_2) = f_2 + xyf_1$$

```
>>> from sympy.polys.distributedmodules import sdm_add
>>> from sympy.polys import lex, QQ
>>> sdm_add([((1, 1, 1), QQ(1))], [((2, 0, 0), QQ(1))], lex, QQ)
[((2, 0, 0), 1), ((1, 1, 1), 1)]
```

$$(xyf_1) + (-xyf_1) = 0'$$

```
>>> sdm_add([((1, 1, 1), QQ(1))], [((1, 1, 1), QQ(-1))], lex, QQ)
[]
```

$$(f_1) + (2f_1) = 3f_1$$

```
>>> sdm_add([((1, 0, 0), QQ(1))], [((1, 0, 0), QQ(2))], lex, QQ)
[((1, 0, 0), 3)]
```

$$(yf_1) + (xf_1) = xf_1 + yf_1$$

```
>>> sdm_add([((1, 0, 1), QQ(1))], [((1, 1, 0), QQ(1))], lex, QQ)
[((1, 1, 0), 1), ((1, 0, 1), 1)]
```

`sympy.polys.distributedmodules.sdm_LM(f)`

Returns the leading monomial of f .

Only valid if $f \neq 0$.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_LM, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(1), (4, 0, 1): QQ(1)}
>>> sdm_LM(sdm_from_dict(dic, lex))
(4, 0, 1)
```

`sympy.polys.distributedmodules.sdm_LT(f)`

Returns the leading term of f .

Only valid if $f \neq 0$.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_LT, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(2), (4, 0, 1): QQ(3)}
>>> sdm_LT(sdm_from_dict(dic, lex))
((4, 0, 1), 3)
```

`sympy.polys.distributedmodules.sdm_mul_term(f, term, O, K)`

Multiply a distributed module element f by a (polynomial) term $term$.

Multiplication of coefficients is done over the ground field K , and monomials are ordered according to O .

Examples

$$0f_1 = 0$$

```
>>> from sympy.polys.distributedmodules import sdm_mul_term
>>> from sympy.polys import lex, QQ
>>> sdm_mul_term([(1, 0, 0), QQ(1)], ((0, 0), QQ(0)), lex, QQ)
[]
```

$$x0 = 0$$

```
>>> sdm_mul_term([], ((1, 0), QQ(1)), lex, QQ)
[]
```

$$(x)(f_1) = xf_1$$

```
>>> sdm_mul_term([(1, 0, 0), QQ(1)], ((1, 0), QQ(1)), lex, QQ)
[((1, 1, 0), 1)]
```

$$(2xy)(3xf_1 + 4yf_2) = 8xy^2f_2 + 6x^2yf_1$$

```
>>> f = [((2, 0, 1), QQ(4)), ((1, 1, 0), QQ(3))]
>>> sdm_mul_term(f, ((1, 1), QQ(2)), lex, QQ)
[((2, 1, 2), 8), ((1, 2, 1), 6)]
```

`sympy.polys.distributedmodules.sdm_zero()`

Return the zero module element.

`sympy.polys.distributedmodules.sdm_deg(f)`

Degree of f .

This is the maximum of the degrees of all its monomials. Invalid if f is zero.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_deg
>>> sdm_deg([(1, 2, 3), 1], ((10, 0, 1), 1), ((2, 3, 4), 4)])
7
```

`sympy.polys.distributedmodules.sdm_from_vector(vec, O, K, **opts)`

Create an sdm from an iterable of expressions.

Coefficients are created in the ground field K , and terms are ordered according to monomial order O . Named arguments are passed on to the polys conversion code and can be used to specify for example generators.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_from_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> sdm_from_vector([x**2+y**2, 2*z], lex, QQ)
[((1, 0, 0, 1), 2), ((0, 2, 0, 0), 1), ((0, 0, 2, 0), 1)]
```

`sympy.polys.distributedmodules.sdm_to_vector(f, gens, K, n=None)`

Convert sdm f into a list of polynomial expressions.

The generators for the polynomial ring are specified via $gens$. The rank of the module is guessed, or passed via n . The ground field is assumed to be K .

Examples

```
>>> from sympy.polys.distributedmodules import sdm_to_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> f = [((1, 0, 0, 1), QQ(2)), ((0, 2, 0, 0), QQ(1)), ((0, 0, 2, 0), QQ(1))]
>>> sdm_to_vector(f, [x, y, z], QQ)
[x**2 + y**2, 2*z]
```

Polynomial factorization algorithms

Many variants of Euclid's algorithm:

Classical remainder sequence

Let K be a field, and consider the ring $K[X]$ of polynomials in a single indeterminate X with coefficients in K . Given two elements f and g of $K[X]$ with $g \neq 0$ there are unique polynomials q

and r such that $f = qg + r$ and $\deg(r) < \deg(g)$ or $r = 0$. They are denoted by $\text{quo}(f, g)$ (quotient) and $\text{rem}(f, g)$ (remainder), so we have the division identity

$$f = \text{quo}(f, g)g + \text{rem}(f, g).$$

It follows that every ideal I of $K[X]$ is a principal ideal, generated by any element $\neq 0$ of minimum degree (assuming I non-zero). In fact, if g is such a polynomial and f is any element of I , $\text{rem}(f, g)$ belongs to I as a linear combination of f and g , hence must be zero; therefore f is a multiple of g .

Using this result it is possible to find a [greatest common divisor](#) (gcd) of any polynomials $f, g, \dots$ in $K[X]$. If I is the ideal formed by all linear combinations of the given polynomials with coefficients in $K[X]$, and d is its generator, then every common divisor of the polynomials also divides d . On the other hand, the given polynomials are multiples of the generator d ; hence d is a gcd of the polynomials, denoted $\text{gcd}(f, g, \dots)$.

An algorithm for the gcd of two polynomials f and g in $K[X]$ can now be obtained as follows. By the division identity, $r = \text{rem}(f, g)$ is in the ideal generated by f and g , as well as f is in the ideal generated by g and r . Hence the ideals generated by the pairs (f, g) and (g, r) are the same. Set $f_0 = f$, $f_1 = g$, and define recursively $f_i = \text{rem}(f_{i-2}, f_{i-1})$ for $i \geq 2$. The recursion ends after a finite number of steps with $f_{k+1} = 0$, since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs (f_{i-1}, f_i) generate the same ideal. In particular, the ideal generated by f and g is generated by f_k alone as $f_{k+1} = 0$. Hence $d = f_k$ is a gcd of f and g .

The sequence of polynomials $f_0, f_1, \dots, f_k$ is called the Euclidean polynomial remainder sequence determined by (f, g) because of the analogy with the classical [Euclidean algorithm](#) for the gcd of natural numbers.

The algorithm may be extended to obtain an expression for d in terms of f and g by using the full division identities to write recursively each f_i as a linear combination of f and g . This leads to an equation

$$d = uf + vg \quad (u, v \in K[X])$$

analogous to [Bézout's identity](#) in the case of integers.

`sympy.polys.euclidtools.dmp_half_gcdex(f, g, u, K)`
Half extended Euclidean algorithm in $F[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

`sympy.polys.euclidtools.dmp_gcdex(f, g, u, K)`
Extended Euclidean algorithm in $F[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

`sympy.polys.euclidtools.dmp_invert(f, g, u, K)`
Compute multiplicative inverse of f modulo g in $F[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x = ring("x", QQ)
```

`sympy.polys.euclidtools.dmp_euclidean_prs(f, g, u, K)`
Euclidean polynomial remainder sequence (PRS) in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

Simplified remainder sequences

Assume, as is usual, that the coefficient field K is the field of fractions of an integral domain A . In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If A is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in $K[X]$ is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial f in $K[X]$ can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation $f = cF$ where $c \in K$ is the content of f and F is a primitive polynomial, i.e., a polynomial in $A[X]$ with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials f and g with their primitive parts. This will only modify $\text{rem}(f, g)$ by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive $\text{gcd}(f, g)$.

This sequence is the primitive polynomial remainder sequence. It is an example of general polynomial remainder sequences where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

`sympy.polys.euclidtools.dmp_primitive_prs(f, g, u, K)`
Primitive polynomial remainder sequence (PRS) in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

Subresultant sequence

The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [Collins67] realized that the so-called subresultant polynomials of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in the subdomain A , so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in A .

To see how subresultants are associated with remainder sequences recall that all polynomials h in the sequence are linear combinations of the given polynomials f and g

$$h = uf + vg$$

with polynomials u and v in $K[X]$. Moreover, as is seen from the extended Euclidean algorithm, the degrees of u and v are relatively low, with limited growth from step to step.

Let $n = \deg(f)$, and $m = \deg(g)$, and assume $n \geq m$. If $\deg(h) = j < m$, the coefficients of the powers X^k ($k > j$) in the products uf and vg cancel each other. In particular, the products must have the same degree, say, l . Then $\deg(u) = l - n$ and $\deg(v) = l - m$ with a total of $2l - n - m + 2$ coefficients to be determined.

On the other hand, the equality $h = uf + vg$ implies that $l - j$ linear combinations of the coefficients are zero, those associated with the powers X^i ($j < i \leq l$), and one has a given non-zero value, namely the leading coefficient of h .

To satisfy these $l - j + 1$ linear equations the total number of coefficients to be determined cannot be lower than $l - j + 1$, in general. This leads to the inequality $l \geq n + m - j - 1$. Taking $l = n + m - j - 1$, we obtain $\deg(u) = m - j - 1$ and $\deg(v) = n - j - 1$.

In the case $j = 0$ the matrix of the resulting system of linear equations is the [Sylvester matrix](#) $S(f, g)$ associated to f and g , an $(n + m) \times (n + m)$ matrix with coefficients of f and g as entries. Its determinant is the [resultant](#) $\text{res}(f, g)$ of the pair (f, g) . It is non-zero if and only if f and g are relatively prime.

For any j in the interval from 0 to m the matrix of the linear system is an $(n + m - 2j) \times (n + m - 2j)$ submatrix of the Sylvester matrix. Its determinant $s_j(f, g)$ is called the j th scalar subresultant of f and g .

If $s_j(f, g)$ is not zero, the associated equation $h = uf + vg$ has a unique solution where $\deg(h) = j$ and the leading coefficient of h has any given value; the one with leading coefficient $s_j(f, g)$ is the j th subresultant polynomial or, briefly, subresultant of the pair (f, g) , and denoted $S_j(f, g)$. This choice guarantees that the remaining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if f and g are in $A[X]$, so is $S_j(f, g)$ as well. This construction of subresultants applies to any j between 0 and m regardless of the value of $s_j(f, g)$; if it is zero, then $\deg(S_j(f, g)) < j$.

The properties of subresultants are as follows. Let $n_0 = \deg(f)$, $n_1 = \deg(g)$, $n_2, \dots, n_k$ be the decreasing sequence of degrees of polynomials in a remainder sequence. Let $0 \leq j \leq n_1$; then

- $s_j(f, g) \neq 0$ if and only if $j = n_i$ for some i .
- $S_j(f, g) \neq 0$ if and only if $j = n_i$ or $j = n_i - 1$ for some i .

Normally, $n_{i-1} - n_i = 1$ for $1 < i \leq k$. If $n_{i-1} - n_i > 1$ for some i (the abnormal case), then $S_{n_{i-1}-1}(f, g)$ and $S_{n_i}(f, g)$ are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the subresultant remainder sequence by setting

$$f_i = S_{n_{i-1}-1}(f, g) \quad (2 \leq i \leq k).$$

In the normal case, these are the same as the $S_{n_i}(f, g)$. He also derived expressions for the constants γ_i in the remainder formulas

$$\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})$$

in terms of the leading coefficients of $f_1, \dots, f_{i-1}$, working in the field K .

Brown and Traub [BrownTraub71] later developed a recursive procedure for computing the coefficients γ_i . Their algorithm deals with elements of the domain A exclusively (assuming $f, g \in A[X]$). However, in the abnormal case there was a problem, a division in A which could only be conjectured to be exact.

This was subsequently justified by Brown [Brown78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of f_i is $s_{n_i-2}(f, g)$ (Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but $s_{n_k}(f, g)$ which is not needed after finding $f_{k+1} = 0$. Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

`sympy.polys.euclidtools.dmp_inner_subresultants(f, g, u, K)`
Subresultant PRS algorithm in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
>>> prs = [f, g, a, b]
>>> sres = [[1], [1], [3, 0, 0, 0, 0], [-3, 0, 0, -12, 1, 0, -54, 8, 729, -216,
↪ 16]]
```

```
>>> R.dmp_inner_subresultants(f, g) == (prs, sres)
True
```

`sympy.polys.euclidtools.dmp_subresultants(f, g, u, K)`
Computes subresultant PRS of two polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
>>> R.dmp_subresultants(f, g) == [f, g, a, b]
True
```

`sympy.polys.euclidtools.dmp_prs_resultant(f, g, u, K)`
Resultant algorithm in $K[X]$ using subresultant PRS.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
>>> res, prs = R.dmp_prs_resultant(f, g)
```

```
>>> res == b           # resultant has n-1 variables
False
>>> res == b.drop(x)
True
>>> prs == [f, g, a, b]
True
```

`sympy.polys.euclidtools.dmp_zz_modular_resultant(f, g, p, u, K)`
Compute resultant of f and g modulo a prime p .

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_modular_resultant(f, g, 5)
-2*y**2 + 1
```

`sympy.polys.euclidtools.dmp_zz_collins_resultant(f, g, u, K)`
Collins's modular resultant algorithm in $\mathbb{Z}[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_collins_resultant(f, g)
-2*y**2 - 5*y + 1
```

`sympy.polys.euclidtools.dmp_qq_collins_resultant(f, g, u, K0)`
Collins's modular resultant algorithm in $\mathbb{Q}[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + y + QQ(2,3)
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_qq_collins_resultant(f, g)
-2*y**2 - 7/3*y + 5/6
```

`sympy.polys.euclidtools.dmp_resultant(f, g, u, K, includePRS=False)`
Computes resultant of two polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> R.dmp_resultant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

`sympy.polys.euclidtools.dmp_discriminant(f, u, K)`
Computes discriminant of a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y,z,t = ring("x,y,z,t", ZZ)
```

```
>>> R.dmp_discriminant(x**2*y + x*z + t)
-4*y*t + z**2
```

`sympy.polys.euclidtools.dmp_rr_prs_gcd(f, g, u, K)`

Computes polynomial GCD using subresultants over a ring.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_rr_prs_gcd(f, g)
(x + y, x + y, x)
```

`sympy.polys.euclidtools.dmp_ff_prs_gcd(f, g, u, K)`

Computes polynomial GCD using subresultants over a field.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x, y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x**2 + x*y + QQ(1,2)*y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_ff_prs_gcd(f, g)
(x + y, 1/2*x + 1/2*y, x)
```

`sympy.polys.euclidtools.dmp_zz_heu_gcd(f, g, u, K)`

Heuristic polynomial GCD in $\mathbb{Z}[X]$.

Given univariate polynomials f and g in $\mathbb{Z}[X]$, returns their GCD and cofactors, i.e. polynomials h , cff and cfg such that:

```
h = gcd(f, g), cff = quo(f, h) and cfg = quo(g, h)
```

The algorithm is purely heuristic which means it may fail to compute the GCD. This will be signaled by raising an exception. In this case you will need to switch to another GCD method.

The algorithm computes the polynomial GCD by evaluating polynomials f and g at certain points and computing (fast) integer GCD of those evaluations. The polynomial GCD is recovered from the integer image by interpolation. The evaluation process reduces f and g variable by variable into a large integer. The final step is to verify if the interpolated polynomial is the correct GCD. This gives cofactors of the input polynomials as a side effect.

References

1. [Liao95] (page 1866)

[Liao95] (page 1866)

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_zz_heu_gcd(f, g)
(x + y, x + y, x)
```

`sympy.polys.euclidtools.dmp_qq_heu_gcd(f, g, u, K0)`

Heuristic polynomial GCD in $Q[X]$.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,4)*x**2 + x*y + y**2
>>> g = QQ(1,2)*x**2 + x*y
```

```
>>> R.dmp_qq_heu_gcd(f, g)
(x + 2*y, 1/4*x + 1/2*y, 1/2*x)
```

`sympy.polys.euclidtools.dmp_inner_gcd(f, g, u, K)`

Computes polynomial GCD and cofactors of f and g in $K[X]$.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_inner_gcd(f, g)
(x + y, x + y, x)
```

`sympy.polys.euclidtools.dmp_gcd(f, g, u, K)`
Computes polynomial GCD of f and g in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_gcd(f, g)
x + y
```

`sympy.polys.euclidtools.dmp_lcm(f, g, u, K)`
Computes polynomial LCM of f and g in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_lcm(f, g)
x**3 + 2*x**2*y + x*y**2
```

`sympy.polys.euclidtools.dmp_content(f, u, K)`
Returns GCD of multivariate coefficients.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_content(2*x*y + 6*x + 4*y + 12)
2*y + 6
```

`sympy.polys.euclidtools.dmp_primitive(f, u, K)`
Returns multivariate content and a primitive polynomial.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_primitive(2*x*y + 6*x + 4*y + 12)
(2*y + 6, x + 2)
```

`sympy.polys.euclidtools.dmp_cancel(f, g, u, K, include=True)`
 Cancel common factors in a rational function f/g .

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```
>>> R.dmp_cancel(2*x**2 - 2, x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

Polynomial factorization in characteristic zero:

`sympy.polys.factortools.dmp_trial_division(f, factors, u, K)`
 Determine multiplicities of factors using trial division.

`sympy.polys.factortools.dmp_zz_mignotte_bound(f, u, K)`
 Mignotte bound for multivariate polynomials in $K[X]$.

`sympy.polys.factortools.dup_zz_hensel_step(m, f, g, h, s, t, K)`
 One step in Hensel lifting in $Z[x]$.

Given positive integer m and $Z[x]$ polynomials f, g, h, s and t such that:

```
f == g*h (mod m)
s*g + t*h == 1 (mod m)

lc(f) is not a zero divisor (mod m)
lc(h) == 1

deg(f) == deg(g) + deg(h)
deg(s) < deg(h)
deg(t) < deg(g)
```

returns polynomials G, H, S and T , such that:

```
f == G*H (mod m**2)
S*G + T*H == 1 (mod m**2)
```

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

`sympy.polys.factortools.dup_zz_hensel_lift(p, f, f_list, l, K)`
 Multifactor Hensel lifting in $Z[x]$.

Given a prime p , polynomial f over $Z[x]$ such that $lc(f)$ is a unit modulo p , monic pair-wise coprime polynomials f_i over $Z[x]$ satisfying:

```
f = lc(f) f_1 ... f_r (mod p)
```

and a positive integer l , returns a list of monic polynomials $F_1, F_2, \dots, F_r$ satisfying:

```
f = lc(f) F_1 ... F_r (mod p**l)
F_i = f_i (mod p), i = 1..r
```

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

`sympy.polys.factortools.dup_zz_zassenhaus(f, K)`
Factor primitive square-free polynomials in $Z[x]$.

`sympy.polys.factortools.dup_zz_irreducible_p(f, K)`
Test irreducibility using Eisenstein's criterion.

`sympy.polys.factortools.dup_cyclotomic_p(f, K, irreducible=False)`
Efficiently test if f is a cyclotomic polynomial.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(f)
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(g)
True
```

`sympy.polys.factortools.dup_zz_cyclotomic_poly(n, K)`
Efficiently generate n -th cyclotomic polynomial.

`sympy.polys.factortools.dup_zz_cyclotomic_factor(f, K)`
Efficiently factor polynomials $x**n - 1$ and $x**n + 1$ in $Z[x]$.

Given a univariate polynomial f in $Z[x]$ returns a list of factors of f , provided that f is in the form $x**n - 1$ or $x**n + 1$ for $n \geq 1$. Otherwise returns None.

Factorization is performed using cyclotomic decomposition of f , which makes this method much faster than any other direct factorization approach (e.g. Zassenhaus's).

References

1. [Weisstein09] (page 1866)

[Weisstein09] (page 1866)

`sympy.polys.factortools.dup_zz_factor_sqf(f, K)`
Factor square-free (non-primitive) polynomials in $Z[x]$.

`sympy.polys.factortools.dup_zz_factor(f, K)`

Factor (non square-free) polynomials in $Z[x]$.

Given a univariate polynomial f in $Z[x]$ computes its complete factorization $f_1, \dots, f_n$ into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Zassenhaus algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial $f = 2 * x ** 4 - 2$:

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_zz_factor(2*x**4 - 2)
(2, [(x - 1, 1), (x + 1, 1), (x**2 + 1, 1)])
```

In result we got the following factorization:

```
f = 2 (x - 1) (x + 1) (x**2 + 1)
```

Note that this is a complete factorization over integers, however over Gaussian integers we can factor the last term.

By default, polynomials $x^{**n}-1$ and $x^{**n}+1$ are factored using cyclotomic decomposition to speedup computations. To disable this behaviour set `cyclotomic=False`.

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

`sympy.polys.factortools.dmp_zz_wang_non_divisors(E, cs, ct, K)`

Wang/EEZ: Compute a set of valid divisors.

`sympy.polys.factortools.dmp_zz_wang_test_points(f, T, ct, A, u, K)`

Wang/EEZ: Test evaluation points for suitability.

`sympy.polys.factortools.dmp_zz_wang_lead_coeffs(f, T, cs, E, H, A, u, K)`

Wang/EEZ: Compute correct leading coefficients.

`sympy.polys.factortools.dmp_zz_diophantine(F, c, A, d, p, u, K)`

Wang/EEZ: Solve multivariate Diophantine equations.

`sympy.polys.factortools.dmp_zz_wang_hensel_lifting(f, H, LC, A, p, u, K)`

Wang/EEZ: Parallel Hensel lifting algorithm.

`sympy.polys.factortools.dmp_zz_wang(f, u, K, mod=None, seed=None)`

Factor primitive square-free polynomials in $Z[X]$.

Given a multivariate polynomial f in $Z[x_1, \dots, x_n]$, which is primitive and square-free in x_1 , computes factorization of f into irreducibles over integers.

The procedure is based on Wang's Enhanced Extended Zassenhaus algorithm. The algorithm works by viewing f as a univariate polynomial in $Z[x_2, \dots, x_n][x_1]$, for which an evaluation mapping is computed:

```
x_2 -> a_2, ..., x_n -> a_n
```

where a_i , for $i = 2, \dots, n$, are carefully chosen integers. The mapping is used to transform f into a univariate polynomial in $Z[x_1]$, which can be factored efficiently using Zassenhaus algorithm. The last step is to lift univariate factors to obtain true multivariate factors. For this purpose a parallel Hensel lifting procedure is used.

The parameter `seed` is passed to `_randint` and can be used to seed `randint` (when an integer) or (for testing purposes) can be a sequence of numbers.

References

1. [Wang78] (page 1866)
2. [Geddes92] (page 1866)

[Wang78] (page 1866), [Geddes92] (page 1866)

`sympy.polys.factortools.dmp_zz_factor(f, u, K)`
Factor (non square-free) polynomials in $Z[X]$.

Given a multivariate polynomial f in $Z[x]$ computes its complete factorization $f_1, \dots, f_n$ into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Enhanced Extended Zassenhaus (EEZ) algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial $f = 2 * (x ** 2 - y ** 2)$:

```
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_zz_factor(2*x**2 - 2*y**2)
(2, [(x - y, 1), (x + y, 1)])
```

In result we got the following factorization:

```
f = 2 (x - y) (x + y)
```

References

1. [Gathen99] (page 1866)

[Gathen99] (page 1866)

`sympy.polys.factortools.dmp_ext_factor(f, u, K)`
 Factor multivariate polynomials over algebraic number fields.

`sympy.polys.factortools.dup_gf_factor(f, K)`
 Factor univariate polynomials over finite fields.

`sympy.polys.factortools.dmp_factor_list(f, u, K0)`
 Factor polynomials into irreducibles in $K[X]$.

`sympy.polys.factortools.dmp_factor_list_include(f, u, K)`
 Factor polynomials into irreducibles in $K[X]$.

`sympy.polys.factortools.dmp_irreducible_p(f, u, K)`
 Returns True if f has no factors over its domain.

Groebner basis algorithms

Groebner bases can be used to answer many problems in computational commutative algebra. Their computation is rather complicated, and very performance-sensitive. We present here various low-level implementations of Groebner basis computation algorithms; please see the previous section of the manual for usage.

`sympy.polys.groebnertools.groebner(seq, ring, method=None)`
 Computes Groebner basis for a set of polynomials in $K[X]$.
 Wrapper around the (default) improved Buchberger and the other algorithms for computing Groebner bases. The choice of algorithm can be changed via `method` argument or `setup()` from `sympy.polys.polyconfig`, where `method` can be either `buchberger` or `f5b`.

`sympy.polys.groebnertools.spoly(p1, p2, ring)`
 Compute $\text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p1)*p1 - \text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p2)*p2$ This is the S-poly provided $p1$ and $p2$ are monic

`sympy.polys.groebnertools.red_groebner(G, ring)`
 Compute reduced Groebner basis, from BeckerWeispfenning93, p. 216
 Selects a subset of generators, that already generate the ideal and computes a reduced Groebner basis for them.

`sympy.polys.groebnertools.is_groebner(G, ring)`
 Check if G is a Groebner basis.

`sympy.polys.groebnertools.is_minimal(G, ring)`
 Checks if G is a minimal Groebner basis.

`sympy.polys.groebnertools.is_reduced(G, ring)`
 Checks if G is a reduced Groebner basis.

`sympy.polys.fglmtools.matrix_fglm(F, ring, O_to)`
 Converts the reduced Groebner basis F of a zero-dimensional ideal w.r.t. O_{from} to a reduced Groebner basis w.r.t. O_{to} .

References

J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering

Groebner basis algorithms for modules are also provided:

`sympy.polys.distributedmodules.sdm_spoly(f, g, O, K, phantom=None)`

Compute the generalized s-polynomial of f and g .

The ground field is assumed to be K , and monomials ordered according to O .

This is invalid if either of f or g is zero.

If the leading terms of f and g involve different basis elements of F , their s-poly is defined to be zero. Otherwise it is a certain linear combination of f and g in which the leading terms cancel. See [SCA, defn 2.3.6] for details.

If `phantom` is not `None`, it should be a pair of module elements on which to perform the same operation(s) as on f and g . The in this case both results are returned.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_spoly
>>> from sympy.polys import QQ, lex
>>> f = [((2, 1, 1), QQ(1)), ((1, 0, 1), QQ(1))]
>>> g = [((2, 3, 0), QQ(1))]
>>> h = [((1, 2, 3), QQ(1))]
>>> sdm_spoly(f, h, lex, QQ)
[]
>>> sdm_spoly(f, g, lex, QQ)
[((1, 2, 1), 1)]
```

`sympy.polys.distributedmodules.sdm_ecart(f)`

Compute the ecart of f .

This is defined to be the difference of the total degree of f and the total degree of the leading monomial of f [SCA, defn 2.3.7].

Invalid if f is zero.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_ecart
>>> sdm_ecart([((1, 2, 3), 1), ((1, 0, 1), 1)])
0
>>> sdm_ecart([((2, 2, 1), 1), ((1, 5, 1), 1)])
3
```

`sympy.polys.distributedmodules.sdm_nf_mora(f, G, O, K, phantom=None)`

Compute a weak normal form of f with respect to G and order O .

The ground field is assumed to be K , and monomials ordered according to O .

Weak normal forms are defined in [SCA, defn 2.3.3]. They are not unique. This function deterministically computes a weak normal form, depending on the order of G .

The most important property of a weak normal form is the following: if R is the ring associated with the monomial ordering (if the ordering is global, we just have $R = K[x_1, \dots, x_n]$, otherwise it is a certain localization thereof), I any ideal of R and G a standard basis for I , then for any $f \in R$, we have $f \in I$ if and only if $NF(f|G) = 0$.

This is the generalized Mora algorithm for computing weak normal forms with respect to arbitrary monomial orders [SCA, algorithm 2.3.9].

If `phantom` is not `None`, it should be a pair of “phantom” arguments on which to perform the same computations as on `f`, `G`, both results are then returned.

`sympy.polys.distributedmodules.sdm_groebner(G, NF, O, K, extended=False)`

Compute a minimal standard basis of `G` with respect to order `O`.

The algorithm uses a normal form `NF`, for example `sdm_nf_mora`. The ground field is assumed to be `K`, and monomials ordered according to `O`.

Let N denote the submodule generated by elements of G . A standard basis for N is a subset S of N , such that $\text{in}(S) = \text{in}(N)$, where for any subset X of F , $\text{in}(X)$ denotes the submodule generated by the initial forms of elements of X . [SCA, defn 2.3.2]

A standard basis is called minimal if no subset of it is a standard basis.

One may show that standard bases are always generating sets.

Minimal standard bases are not unique. This algorithm computes a deterministic result, depending on the particular order of G .

If `extended=True`, also compute the transition matrix from the initial generators to the groebner basis. That is, return a list of coefficient vectors, expressing the elements of the groebner basis in terms of the elements of G .

This functions implements the “sugar” strategy, see

Giovini et al: “One sugar cube, please” OR Selection strategies in Buchberger algorithm.

Exceptions

These are exceptions defined by the polynomials module.

TODO sort and explain

class `sympy.polys.polyerrors.BasePolynomialError`

Base class for polynomial related exceptions.

class `sympy.polys.polyerrors.ExactQuotientFailed(f, g, dom=None)`

class `sympy.polys.polyerrors.OperationNotSupported(poly, func)`

class `sympy.polys.polyerrors.HeuristicGCDFailed`

class `sympy.polys.polyerrors.HomomorphismFailed`

class `sympy.polys.polyerrors.IsomorphismFailed`

class `sympy.polys.polyerrors.ExtraneousFactors`

class `sympy.polys.polyerrors.EvaluationFailed`

class `sympy.polys.polyerrors.RefinementFailed`

class `sympy.polys.polyerrors.CoercionFailed`

class `sympy.polys.polyerrors.NotInvertible`

class `sympy.polys.polyerrors.NotReversible`

class `sympy.polys.polyerrors.NotAlgebraic`

class `sympy.polys.polyerrors.DomainError`

class `sympy.polys.polyerrors.PolynomialError`

class `sympy.polys.polyerrors.UnificationFailed`

class `sympy.polys.polyerrors.GeneratorsNeeded`

```
class sympy.polys.polyerrors.ComputationFailed(func, nargs, exc)
class sympy.polys.polyerrors.GeneratorsError
class sympy.polys.polyerrors.UnivariatePolynomialError
class sympy.polys.polyerrors.MultivariatePolynomialError
class sympy.polys.polyerrors.PolificationFailed(opt, origs, exprs, seq=False)
class sympy.polys.polyerrors.OptionError
class sympy.polys.polyerrors.FlagError
```

Reference

Modular GCD

`sympy.polys.modulargcd.modgcd_univariate(f, g)`

Computes the GCD of two polynomials in $\mathbb{Z}[x]$ using a modular algorithm.

The algorithm computes the GCD of two univariate integer polynomials f and g by computing the GCD in $\mathbb{Z}_p[x]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. Trial division is only made for candidates which are very likely the desired GCD.

Parameters f : PolyElement

univariate integer polynomial

g : PolyElement

univariate integer polynomial

Returns h : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

References

1. [Monagan00] (page 1867)

[Monagan00] (page 1867)

Examples

```
>>> from sympy.polys.modulargcd import modgcd_univariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**5 - 1
>>> g = x - 1
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(x - 1, x**4 + x**3 + x**2 + x + 1, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = 6*x**2 - 6
>>> g = 2*x**2 + 4*x + 2
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(2*x + 2, 3*x - 3, x + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd.modgcd_bivariate(f, g)`

Computes the GCD of two polynomials in $\mathbb{Z}[x, y]$ using a modular algorithm.

The algorithm computes the GCD of two bivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x, y]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the bivariate GCD over $\mathbb{Z}_p$, the polynomials $f \bmod p$ and $g \bmod p$ are evaluated at $y = a$ for certain $a \in \mathbb{Z}_p$ and then their univariate GCD in $\mathbb{Z}_p[x]$ is computed. Interpolating those yields the bivariate GCD in $\mathbb{Z}_p[x, y]$. To verify the result in $\mathbb{Z}[x, y]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters f : PolyElement

bivariate integer polynomial

g : PolyElement

bivariate integer polynomial

Returns h : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

References

1. [Monagan00] (page 1867)

[Monagan00] (page 1867)

Examples

```
>>> from sympy.polys.modulargcd import modgcd_bivariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x**2*y - x**2 - 4*y + 4
>>> g = x + 2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + 2, x*y - x - 2*y + 2, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd.modgcd_multivariate(f, g)`

Compute the GCD of two polynomials in $\mathbb{Z}[x_0, \dots, x_{k-1}]$ using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over $\mathbb{Z}_p$ the recursive subroutine `_modgcd_multivariate_p` is used. To verify the result in $\mathbb{Z}[x_0, \dots, x_{k-1}]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters **f** : PolyElement

multivariate integer polynomial

g : PolyElement

multivariate integer polynomial

Returns **h** : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement
 cofactor of g , i.e. $\frac{g}{h}$

See also:

`_modgcd_multivariate_p`

References

1. [Monagan00] (page 1867)
2. [Brown71] (page 1867)

[Monagan00] (page 1867), [Brown71] (page 1867)

Examples

```
>>> from sympy.polys.modulargcd import modgcd_multivariate
>>> from sympy.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

```
>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y, z = ring("x, y, z", ZZ)
```

```
>>> f = x*z**2 - y*z**2
>>> g = x**2*z + z
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(z, x*z - y*z, x**2 + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd.func_field_modgcd(f, g)`

Compute the GCD of two polynomials f and g in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$ using a modular algorithm.

The algorithm first computes the primitive associate $\check{m}_\alpha(z)$ of the minimal polynomial m_α in $\mathbb{Z}[z]$ and the primitive associates of f and g in $\mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\check{m}_\alpha)[x_0]$. Then it computes the GCD in $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$. This is done by calculating the GCD in $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstruction. The GCD over $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ is computed with a recursive subroutine, which evaluates the polynomials at $x_{n-1} = a$ for suitable evaluation points $a \in \mathbb{Z}_p$ and then calls itself recursively until the ground domain does no longer contain any parameters. For $\mathbb{Z}_p[z]/(\check{m}_\alpha(z))[x_0]$ the Euclidean Algorithm is used. The results of those recursive calls are then interpolated and Rational Function Reconstruction is used to obtain the correct coefficients. The results, both in $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$ and $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$, are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in $\mathbb{Q}(\alpha)[x_1, \dots, x_{n-1}]$ have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this `func_field_modgcd` is called recursively.

Parameters **f, g** : PolyElement

polynomials in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

Returns **h** : PolyElement

monic GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

References

1. [Hoeij04] (page 1867)

[Hoeij04] (page 1867)

Examples

```
>>> from sympy.polys.modulargcd import func_field_modgcd
>>> from sympy.polys import AlgebraicField, QQ, ring
>>> from sympy import sqrt
```

```
>>> A = AlgebraicField(QQ, sqrt(2))
>>> R, x = ring('x', A)
```

```
>>> f = x**2 - 2
>>> g = x + sqrt(2)
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)
True
>>> cff * h == f
```

(continues on next page)

(continued from previous page)

```
True
>>> cfg * h == g
True
```

```
>>> R, x, y = ring('x, y', A)
```

```
>>> f = x**2 + 2*sqrt(2)*x*y + 2*y**2
>>> g = x + sqrt(2)*y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)*y
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x + sqrt(2)*y
>>> g = x + y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == R.one
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

Series Manipulation using Polynomials

Any finite Taylor series, for all practical purposes is, in fact a polynomial. This module makes use of the efficient representation and operations of sparse polynomials for very fast multivariate series manipulations. Typical speedups compared to SymPy's `series` method are in the range 20-100, with the gap widening as the series being handled gets larger.

All the functions expand any given series on some ring specified by the user. Thus, the coefficients of the calculated series depend on the ring being used. For example:

```
>>> from sympy.polys import ring, QQ, RR
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x*y, x, 5)
-1/6*x**3*y**3 + x*y
```

QQ stands for the Rational domain. Here all coefficients are rationals. It is recommended to use QQ with ring series as it automatically chooses the fastest Rational type.

Similarly, if a Real domain is used:

```
>>> R, x, y = ring('x, y', RR)
>>> rs_sin(x*y, x, 5)
-0.1666666666666667*x**3*y**3 + x*y
```

Though the definition of a polynomial limits the use of Polynomial module to Taylor series, we extend it to allow Laurent and even Puiseux series (with fractional exponents):

```
>>> from sympy.polys.ring_series import rs_cos, rs_tan
>>> R, x, y = ring('x, y', QQ)

>>> rs_cos(x + x*y, x, 3)/x**3
-1/2*x**(-1)*y**2 - x**(-1)*y - 1/2*x**(-1) + x**(-3)

>>> rs_tan(x**QQ(2, 5)*y**QQ(1, 2), x, 2)
1/3*x**(6/5)*y**(3/2) + x**(2/5)*y**(1/2)
```

By default, PolyElement did not allow non-natural numbers as exponents. It converted a fraction to an integer and raised an error on getting negative exponents. The goal of the ring series module is fast series expansion, and not to use the polys module. The reason we use it as our backend is simply because it implements a sparse representation and most of the basic functions that we need. However, this default behaviour of polys was limiting for ring series.

Note that there is no such constraint (in having rational exponents) in the data-structure used by polys- dict. Sparse polynomials (PolyElement) use the Python dict to store a polynomial term by term, where a tuple of exponents is the key and the coefficient of that term is the value. There is no reason why we can't have rational values in the dict so as to support rational exponents.

So the approach we took was to modify sparse polys to allow non-natural exponents. And it turned out to be quite simple. We only had to delete the conversion to int of exponents in the `__pow__` method of PolyElement. So:

```
>>> x**QQ(3, 4)
x**(3/4)
```

and not 1 as was the case earlier.

Though this change violates the definition of a polynomial, it doesn't break anything yet. Ideally, we shouldn't modify polys in any way. But to have all the series capabilities we want, no other simple way was found. If need be, we can separate the modified part of polys from core polys. It would be great if any other elegant solution is found.

All series returned by the functions of this module are instances of the PolyElement class. To use them with other SymPy types, convert them to Expr:

```
>>> from sympy.polys.ring_series import rs_exp
>>> from sympy.abc import a, b, c
>>> series = rs_exp(x, x, 5)
>>> a + series.as_expr()
a + x**4/24 + x**3/6 + x**2/2 + x + 1
```

rs_series

Direct use of elementary ring series functions does give more control, but is limiting at the same time. Creating an appropriate ring for the desired series expansion and knowing which ring series function to call, are things not everyone might be familiar with.

rs_series is a function that takes an arbitrary Expr and returns its expansion by calling the appropriate ring series functions. The returned series is a polynomial over the simplest (almost) possible ring that does the job. It recursively builds the ring as it parses the given expression, adding generators to the ring when it needs them. Some examples:

```
>>> rs_series(sin(a + b), a, 5)
1/24*sin(b)*a**4 - 1/2*sin(b)*a**2 + sin(b) - 1/6*cos(b)*a**3 + cos(b)*a

>>> rs_series(sin(exp(a*b) + cos(a + c)), a, 2)
-sin(c)*cos(cos(c) + 1)*a + cos(cos(c) + 1)*a*b + sin(cos(c) + 1)

>>> rs_series(sin(a + b)*cos(a + c)*tan(a**2 + b), a, 2)
cos(b)*cos(c)*tan(b)*a - sin(b)*sin(c)*tan(b)*a + sin(b)*cos(c)*tan(b)
```

It can expand complicated multivariate expressions involving multiple functions and most importantly, it does so blazingly fast:

```
>>> %timeit ((sin(a) + cos(a))**10).series(a, 0, 5)
1 loops, best of 3: 1.33 s per loop

>>> %timeit rs_series((sin(a) + cos(a))**10, a, 5)
100 loops, best of 3: 4.13 ms per loop
```

rs_series is over 300 times faster. Given an expression to expand, there is some fixed overhead to parse it. Thus, for larger orders, the speed improvement becomes more prominent:

```
>>> %timeit rs_series((sin(a) + cos(a))**10, a, 100)
10 loops, best of 3: 32.8 ms per loop
```

To figure out the right ring for a given expression, *rs_series* uses the *sring* function, which in turn uses other functions of polys. As explained above, non-natural exponents are not allowed. But the restriction is on exponents and not generators. So, polys allows all sorts of symbolic terms as generators to make sure that the exponent is a natural number:

```
>>> from sympy.polys.rings import sring
>>> R, expr = sring(1/a**3 + a**QQ(3, 7)); R
Polynomial ring in 1/a, a**(1/7) over ZZ with lex order
```

In the above example, $1/a$ and $a^{1/7}$ will be treated as completely different atoms. For all practical purposes, we could let $b = 1/a$ and $c = a^{1/7}$ and do the manipulations. Effectively, expressions involving $1/a$ and $a^{1/7}$ (and their powers) will never simplify:

```
>>> expr*R(1/a)
(1/a)**2 + (1/a)*(a**(1/7))**3
```

This leads to similar issues with manipulating Laurent and Puiseux series as faced earlier. Fortunately, this time we have an elegant solution and are able to isolate the series and polys behaviour from one another. We introduce a boolean flag *series* in the list of allowed Options for polynomials (see `sympy.polys.polyoptions.Options`). Thus, when we want *sring* to allow rational exponents we supply a *series=True* flag to *sring*:

```
>>> rs_series(sin(a**QQ(1, 2)), a, 3)
-1/5040*a**(7/3) + 1/120*a**(5/3) - 1/6*a + a**(1/3)
```

Contribute

rs_series is not fully implemented yet. As of now, it supports only multivariate Taylor expansions of expressions involving `sin`, `cos`, `exp` and `tan`. Adding the remaining functions is not at all difficult and they will be gradually added. If you are interested in helping, read the comments in `ring_series.py`. Currently, it does not support Puiseux series (though the elementary functions do). This is expected to be fixed soon.

You can also add more functions to `ring_series.py`. Only elementary functions are supported currently. The long term goal is to replace SymPy's current series method with *rs_series*.

Manipulation of power series

Functions in this module carry the prefix `rs_`, standing for “ring series”. They manipulate finite power series in the sparse representation provided by `polys.ring.ring`.

Elementary functions

`sympy.polys.ring_series.rs_log(p, x, prec)`
The Logarithm of p modulo $O(x^{**prec})$.

Notes

Truncation of integral $\int dx \, x^{p-1} \frac{dp}{dx}$ is used.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_log
>>> R, x = ring('x', QQ)
>>> rs_log(1 + x, x, 8)
1/7*x**7 - 1/6*x**6 + 1/5*x**5 - 1/4*x**4 + 1/3*x**3 - 1/2*x**2 + x
>>> rs_log(x**QQ(3, 2) + 1, x, 5)
1/3*x**(9/2) - 1/2*x**3 + x**(3/2)
```

`sympy.polys.ring_series.rs_LambertW(p, x, prec)`
Calculate the series expansion of the principal branch of the Lambert W function.

See also:

`LambertW`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_LambertW
>>> R, x, y = ring('x, y', QQ)
>>> rs_LambertW(x + x*y, x, 3)
-x**2*y**2 - 2*x**2*y - x**2 + x*y + x
```

`sympy.polys.ring_series.rs_exp(p, x, prec)`
Exponentiation of a series modulo $O(x^{**prec})$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_exp
>>> R, x = ring('x', QQ)
>>> rs_exp(x**2, x, 7)
1/6*x**6 + 1/2*x**4 + x**2 + 1
```

`sympy.polys.ring_series.rs_atan(p, x, prec)`
The arctangent of a series
Return the series expansion of the atan of p, about 0.

See also:

atan

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atan
>>> R, x, y = ring('x, y', QQ)
>>> rs_atan(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

`sympy.polys.ring_series.rs_asin(p, x, prec)`
Arcsine of a series
Return the series expansion of the asin of p, about 0.

See also:

asin

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_asin
>>> R, x, y = ring('x, y', QQ)
>>> rs_asin(x, x, 8)
5/112*x**7 + 3/40*x**5 + 1/6*x**3 + x
```

`sympy.polys.ring_series.rs_tan(p, x, prec)`

Tangent of a series.

Return the series expansion of the tan of p, about 0.

See also:

`_tan1`, `tan`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tan
>>> R, x, y = ring('x, y', QQ)
>>> rs_tan(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

`sympy.polys.ring_series.rs_cot(p, x, prec)`

Cotangent of a series

Return the series expansion of the cot of p, about 0.

See also:

`cot`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cot
>>> R, x, y = ring('x, y', QQ)
>>> rs_cot(x, x, 6)
-2/945*x**5 - 1/45*x**3 - 1/3*x + x**(-1)
```

`sympy.polys.ring_series.rs_sin(p, x, prec)`

Sine of a series

Return the series expansion of the sin of p, about 0.

See also:

`sin`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x + x*y, x, 4)
-1/6*x**3*y**3 - 1/2*x**3*y**2 - 1/2*x**3*y - 1/6*x**3 + x*y + x
>>> rs_sin(x**QQ(3, 2) + x*y**QQ(7, 5), x, 4)
-1/2*x**(7/2)*y**(14/5) - 1/6*x**3*y**(21/5) + x**(3/2) + x*y**(7/5)
```


`sympy.polys.ring_series.rs_cos(p, x, prec)`

Cosine of a series

Return the series expansion of the cos of p, about 0.

See also:

`cos`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cos
>>> R, x, y = ring('x, y', QQ)
>>> rs_cos(x + x*y, x, 4)
-1/2*x**2*y**2 - x**2*y - 1/2*x**2 + 1
>>> rs_cos(x + x*y, x, 4)/x**QQ(7, 5)
-1/2*x**(3/5)*y**2 - x**(3/5)*y - 1/2*x**(3/5) + x**(-7/5)
```

`sympy.polys.ring_series.rs_cos_sin(p, x, prec)`

Return the tuple $(rs\_cos(p, x, prec), rs\_sin(p, x, prec))$.

Is faster than calling `rs_cos` and `rs_sin` separately

`sympy.polys.ring_series.rs_atanh(p, x, prec)`

Hyperbolic arctangent of a series

Return the series expansion of the atanh of p, about 0.

See also:

`atanh`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_atanh(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

`sympy.polys.ring_series.rs_sinh(p, x, prec)`

Hyperbolic sine of a series

Return the series expansion of the sinh of p, about 0.

See also:

`sinh`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sinh
>>> R, x, y = ring('x, y', QQ)
>>> rs_sinh(x + x*y, x, 4)
1/6*x**3*y**3 + 1/2*x**3*y**2 + 1/2*x**3*y + 1/6*x**3 + x*y + x
```

`sympy.polys.ring_series.rs_cosh(p, x, prec)`

Hyperbolic cosine of a series

Return the series expansion of the cosh of p, about 0.

See also:

`cosh`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cosh
>>> R, x, y = ring('x, y', QQ)
>>> rs_cosh(x + x*y, x, 4)
1/2*x**2*y**2 + x**2*y + 1/2*x**2 + 1
```

`sympy.polys.ring_series.rs_tanh(p, x, prec)`

Hyperbolic tangent of a series

Return the series expansion of the tanh of p, about 0.

See also:

`tanh`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_tanh(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

`sympy.polys.ring_series.rs_hadamard_exp(p1, inverse=False)`

Return sum $f_i/i!x^{**i}$ from sum $f_i x^{**i}$, where x is the first variable.

If `inverse=True` return sum $f_i i! x^{**i}$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_hadamard_exp
>>> R, x = ring('x', QQ)
```

(continues on next page)

(continued from previous page)

```
>>> p = 1 + x + x**2 + x**3
>>> rs_hadamard_exp(p)
1/6*x**3 + 1/2*x**2 + x + 1
```

Operations

`sympy.polys.ring_series.rs_mul(p1, p2, x, prec)`

Return the product of the given two series, modulo $O(x^{**prec})$.

`x` is the series variable or its position in the generators.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_mul
>>> R, x = ring('x', QQ)
>>> p1 = x**2 + 2*x + 1
>>> p2 = x + 1
>>> rs_mul(p1, p2, x, 3)
3*x**2 + 3*x + 1
```

`sympy.polys.ring_series.rs_square(p1, x, prec)`

Square the series modulo $O(x^{**prec})$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_square
>>> R, x = ring('x', QQ)
>>> p = x**2 + 2*x + 1
>>> rs_square(p, x, 3)
6*x**2 + 4*x + 1
```

`sympy.polys.ring_series.rs_pow(p1, n, x, prec)`

Return $p1^{**n}$ modulo $O(x^{**prec})$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_pow
>>> R, x = ring('x', QQ)
>>> p = x + 1
>>> rs_pow(p, 4, x, 3)
6*x**2 + 4*x + 1
```

`sympy.polys.ring_series.rs_series_inversion(p, x, prec)`

Multivariate series inversion $1/p$ modulo $O(x^{**prec})$.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_inversion
>>> R, x, y = ring('x, y', QQ)
>>> rs_series_inversion(1 + x*y**2, x, 4)
-x**3*y**6 + x**2*y**4 - x*y**2 + 1
>>> rs_series_inversion(1 + x*y**2, y, 4)
-x*y**2 + 1
>>> rs_series_inversion(x + x**2, x, 4)
x**3 - x**2 + x - 1 + x**(-1)
```

`sympy.polys.ring_series.rs_series_reversion(p, x, n, y)`

Reversion of a series.

p is a series with $O(x^{**n})$ of the form $p = a * x + f(x)$ where a is a number different from 0.

$f(x) = \text{sum}(a\_k * x\_k, \text{kinrange}(2, n))$

a_k : Can depend polynomially on other variables, not indicated. x : Variable with name x. y : Variable with name y.

Solve $p = y$, that is, given $a * x + f(x) - y = 0$, find the solution $x = r(y)$ up to $O(y^{**n})$

Algorithm:

If r_i is the solution at order i , then: $a * r\_i + f(r\_i) - y = O(y^{**i+1})$

and if $r\_(i + 1)$ is the solution at order $i + 1$, then: $a * r\_(i + 1) + f(r\_(i + 1)) - y = O(y^{**i+2})$

We have, $r\_(i + 1) = r\_i + e$, such that, $a * e + f(r\_i) = O(y^{**i+2})$ or $e = -f(r\_i)/a$

So we use the recursion relation: $r\_(i + 1) = r\_i - f(r\_i)/a$ with the boundary condition: $r\_1 = y$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_reversion, rs_trunc
>>> R, x, y, a, b = ring('x, y, a, b', QQ)
>>> p = x - x**2 - 2*b*x**2 + 2*a*b*x**2
>>> p1 = rs_series_reversion(p, x, 3, y); p1
-2*y**2*a*b + 2*y**2*b + y**2 + y
>>> rs_trunc(p.compose(x, p1), y, 3)
y
```

`sympy.polys.ring_series.rs_nth_root(p, n, x, prec)`

Multivariate series expansion of the n th root of p .

Parameters n : $p^{**1/n}$ is returned.

x : PolyElement

prec : Order of the expanded series.

Notes

The result of this function is dependent on the ring over which the polynomial has been defined. If the answer involves a root of a constant, make sure that the polynomial is over a real field. It can not yet handle roots of symbols.

Examples

```
>>> from sympy.polys.domains import QQ, RR
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_nth_root
>>> R, x, y = ring('x, y', QQ)
>>> rs_nth_root(1 + x + x*y, -3, x, 3)
2/9*x**2*y**2 + 4/9*x**2*y + 2/9*x**2 - 1/3*x*y - 1/3*x + 1
>>> R, x, y = ring('x, y', RR)
>>> rs_nth_root(3 + x + x*y, 3, x, 2)
0.160249952256379*x*y + 0.160249952256379*x + 1.44224957030741
```

`sympy.polys.ring_series.rs_trunc(p1, x, prec)`

Truncate the series in the x variable with precision prec, that is, modulo $O(x^{**prec})$

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 12)
x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 10)
x**5 + x + 1
```

`sympy.polys.ring_series.rs_subs(p, rules, x, prec)`

Substitution with truncation according to the mapping in rules.

Return a series with precision prec in the generator x

Note that substitutions are not done one after the other

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)
>>> p = x**2 + y**2
>>> rs_subs(p, {x: x+y, y: x+2*y}, x, 3)
2*x**2 + 6*x*y + 5*y**2
>>> (x + y)**2 + (x + 2*y)**2
2*x**2 + 6*x*y + 5*y**2
```

which differs from

```
>>> rs_subs(rs_subs(p, {x: x+ y}, x, 3), {y: x+ 2*y}, x, 3)
5*x**2 + 12*x*y + 8*y**2
```

Parameters **p** : PolyElement Input series.

rules : dict with substitution mappings.

x : PolyElement in which the series truncation is to be done.

prec : Integer order of the series after truncation.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)
>>> rs_subs(x**2+y**2, {y: (x+y)**2}, x, 3)
6*x**2*y**2 + x**2 + 4*x*y**3 + y**4
```

`sympy.polys.ring_series.rs_diff(p, x)`

Return partial derivative of p with respect to x.

Parameters **x** : PolyElement with respect to which p is differentiated.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_diff
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_diff(p, x)
2*x*y**3 + 1
```

`sympy.polys.ring_series.rs_integrate(p, x)`

Integrate p with respect to x.

Parameters **x** : PolyElement with respect to which p is integrated.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_integrate
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_integrate(p, x)
1/3*x**3*y**3 + 1/2*x**2
```

`sympy.polys.ring_series.rs_newton(p, x, prec)`

Compute the truncated Newton sum of the polynomial p

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_newton
>>> R, x = ring('x', QQ)
>>> p = x**2 - 2
>>> rs_newton(p, x, 5)
8*x**4 + 4*x**2 + 2
```

`sympy.polys.ring_series.rs_compose_add(p1, p2)`
 compute the composed sum $\text{prod}(p2(x - \text{beta})$ for beta root of p1)

References

A. Bostan, P. Flajolet, B. Salvy and E. Schost “Fast Computation with Two Algebraic Numbers”, (2002) Research Report 4579, Institut National de Recherche en Informatique et en Automatique

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_compose_add
>>> R, x = ring('x', QQ)
>>> f = x**2 - 2
>>> g = x**2 - 3
>>> rs_compose_add(f, g)
x**4 - 10*x**2 + 1
```

Utility functions

`sympy.polys.ring_series.rs_is_puiseux(p, x)`
 Test if p is Puiseux series in x.
 Raise an exception if it has a negative power in x.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_is_puiseux
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_is_puiseux(p, x)
True
```

`sympy.polys.ring_series.rs_puiseux(f, p, x, prec)`
 Return the puiseux series for $f(p, x, \text{prec})$.
 To be used when function f is implemented only for regular series.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_puiseux, rs_exp
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_puiseux(rs_exp, p, x, 1)
1/2*x**(4/5) + x**(2/3) + x**(2/5) + 1
```

`sympy.polys.ring_series.rs_puiseux2(f, p, q, x, prec)`

Return the puiseux series for $f(p, q, x, prec)$.

To be used when function `f` is implemented only for regular series.

`sympy.polys.ring_series.rs_series_from_list(p, c, x, prec, concur=1)`

Return a series $sumc[n] * p ** n$ modulo $O(x ** prec)$.

It reduces the number of multiplications by summing concurrently.

$ax = [1, p, p ** 2, \dots, p ** (J - 1)]$ $s = sum(c[i] * ax[i] \text{ for } i \text{ in } range(r, (r + 1) * J)) * p ** ((K - 1) * J)$
with $K \geq (n + 1) / J$

See also:

`sympy.polys.ring.compose`

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_from_list, rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**2 + x + 1
>>> c = [1, 2, 3]
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> rs_trunc(1 + 2*p + 3*p**2, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> pc = R.from_list(list(reversed(c)))
>>> rs_trunc(pc.compose(x, p), x, 4)
6*x**3 + 11*x**2 + 8*x + 6
```

`sympy.polys.ring_series.rs_fun(p, f, *args)`

Function of a multivariate series computed by substitution.

The case with `f` method name is used to compute rs_tan and rs_nth_root of a multivariate series:

$rs\_fun(p, tan, iv, prec)$

$\tan$ series is first computed for a dummy variable $\_x$, i.e., $rs\_tan(\_x, iv, prec)$. Then we substitute $\_x$ with p to get the desired series

Parameters **p** : PolyElement The multivariate series to be expanded.

f : *ring_series* function to be applied on p .

args[-2] : PolyElement with respect to which, the series is to be expanded.

args[-1] : Required order of the expanded series.

Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_fun, _tan1
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x*y + x**2*y + x**3*y**2
>>> rs_fun(p, _tan1, x, 4)
1/3*x**3*y**3 + 2*x**3*y**2 + x**3*y + 1/3*x**3 + x**2*y + x*y + x
```

`sympy.polys.ring_series.mul_xin(p, i, n)`

Return $p * x_i * \dots * x_i$.

x_i is the i th variable in p .

`sympy.polys.ring_series.pow_xin(p, i, n)`

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import pow_xin
>>> R, x, y = ring('x, y', QQ)
>>> p = x**QQ(2,5) + x + x**QQ(2,3)
>>> index = p.ring.gens.index(x)
>>> pow_xin(p, index, 15)
x**15 + x**10 + x**6
```

Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

5.19 Printing System

See the [Printing](#) (page 16) section in Tutorial for introduction into printing.

This guide documents the printing system in SymPy and how it works internally.

5.19.1 Printer Class

Printing subsystem driver

SymPy's printing system works the following way: Any expression can be passed to a designated Printer who then is responsible to return an adequate representation of that expression.

The basic concept is the following:

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the `emptyPrinter` method for the printer.

Which Method is Responsible for Printing?

The whole printing process is started by calling `.doprint(expr)` on the printer which you want to use. This method looks for an appropriate method which can print the given expression in the given style that the printer defines. While looking for the method, it follows these steps:

1. **Let the object print itself if it knows how.**

The printer looks for a specific method in every object. The name of that method depends on the specific printer and is defined under `Printer.printmethod`. For example, `StrPrinter` calls `_sympystr` and `LatexPrinter` calls `_latex`. Look at the documentation of the printer that you want to use. The name of the method is specified there.

This was the original way of doing printing in sympy. Every class had its own `latex`, `mathml`, `str` and `repr` methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefore all printing code was combined into the different printers, which works great for built-in sympy objects, but not that good for user defined classes where it is inconvenient to patch the printers.

2. **Take the best fitting method defined in the printer.**

The printer loops through `expr` classes (class + its bases), and tries to dispatch the work to `_print_<EXPR_CLASS>`

e.g., suppose we have the following class hierarchy:

```
Basic
|
Atom
|
Number
|
Rational
```

then, for `expr=Rational(...)`, the Printer will try to call printer methods in the order as shown in the figure below:

```
p._print(expr)
|
|-- p._print_Rational(expr)
|
|-- p._print_Number(expr)
|
|-- p._print_Atom(expr)
|
'-- p._print_Basic(expr)
```

if `._print_Rational` method exists in the printer, then it is called, and the result is returned back. Otherwise, the printer tries to call `._print_Number` and so on.

3. **As a fall-back use the `emptyPrinter` method for the printer.**

As fall-back `self.emptyPrinter` will be called with the expression. If not defined in the Printer subclass this will be the same as `str(expr)`.

Example of Custom Printer

In the example below, we have a printer which prints the derivative of a function in a shorter form.

```
from sympy import Symbol
from sympy.printing.latex import LatexPrinter, print_latex
from sympy.core.function import UndefinedFunction, Function

class MyLatexPrinter(LatexPrinter):
    """Print derivative of a function of symbols in a shorter form.
    """
    def _print_Derivative(self, expr):
        function, *vars = expr.args
        if not isinstance(type(function), UndefinedFunction) or \
            not all(isinstance(i, Symbol) for i in vars):
            return super()._print_Derivative(expr)

        # If you want the printer to work correctly for nested
        # expressions then use self._print() instead of str() or latex().
        # See the example of nested modulo below in the custom printing
        # method section.
        return "{}_{{{}}}".format(
            self._print(Symbol(function.func.__name__)),
            ''.join(self._print(i) for i in vars))

def print_my_latex(expr):
    """ Most of the printers define their own wrappers for print().
    These wrappers usually take printer settings. Our printer does not have
    any settings.
    """
    print(MyLatexPrinter().doprint(expr))

y = Symbol("y")
x = Symbol("x")
f = Function("f")
expr = f(x, y).diff(x, y)

# Print the expression using the normal latex printer and our custom
# printer.
print_latex(expr)
print_my_latex(expr)
```

The output of the code above is:

```
\frac{\partial^2}{\partial x \partial y} f(\left (x,y \right ))
f_{xy}
```

Example of Custom Printing Method

In the example below, the latex printing of the modulo operator is modified. This is done by overriding the method `_latex` of `Mod`.

```
from sympy import Symbol, Mod, Integer
from sympy.printing.latex import print_latex

class ModOp(Mod):
    def _latex(self, printer=None):
        # Always use printer.doprint() otherwise nested expressions won't
        # work. See the example of ModOpWrong.
        a, b = [printer.doprint(i) for i in self.args]
        return r"\operatorname{Mod}{\left( %s,%s \right)}" % (a,b)

class ModOpWrong(Mod):
    def _latex(self, printer=None):
        a, b = [str(i) for i in self.args]
        return r"\operatorname{Mod}{\left( %s,%s \right)}" % (a,b)

x = Symbol('x')
m = Symbol('m')

print_latex(ModOp(x, m))
print_latex(Mod(x, m))

# Nested modulo.
print_latex(ModOp(ModOp(x, m), Integer(7)))
print_latex(ModOpWrong(ModOpWrong(x, m), Integer(7)))
```

The output of the code above is:

```
\operatorname{Mod}{\left( x,m \right)}
x\bmod{m}
\operatorname{Mod}{\left( \operatorname{Mod}{\left( x,m \right)},7 \right)}
\operatorname{Mod}{\left( \operatorname{ModOpWrong}(x, m),7 \right)}
```

The main class responsible for printing is `Printer` (see also its [source code](#)):

class sympy.printing.printer.**Printer**(settings=None)
Generic printer

Its job is to provide infrastructure for implementing new printers easily.

If you want to define your custom `Printer` or your custom printing method for your custom class then see the example above: [printer_example](#) (page 1021) .

Attributes

printmethod	

printmethod = None

_print(expr, **kwargs)
Internal dispatcher

Tries the following concepts to print an expression:

1. Let the object print itself if it knows how.

2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

doprint(expr)

Returns printer's representation for expr (as a string)

classmethod set_global_settings(**settings)

Set system-wide printing settings.

5.19.2 PrettyPrinter Class

The pretty printing subsystem is implemented in `sympy.printing.pretty.pretty` by the `PrettyPrinter` class deriving from `Printer`. It relies on the modules `sympy.printing.pretty.stringPict`, and `sympy.printing.pretty.pretty_symbology` for rendering nice-looking formulas.

The module `stringPict` provides a base class `stringPict` and a derived class `prettyForm` that ease the creation and manipulation of formulas that span across multiple lines.

The module `pretty_symbology` provides primitives to construct 2D shapes (`hline`, `vline`, etc) together with a technique to use unicode automatically when possible.

class `sympy.printing.pretty.pretty.PrettyPrinter`(settings=None)

Printer, which converts an expression into 2D ASCII-art figure.

printmethod = `'_pretty'`

`sympy.printing.pretty.pretty.pretty`(expr, **settings)

Returns a string containing the prettified form of expr.

For information on keyword arguments see `pretty_print` function.

`sympy.printing.pretty.pretty.pretty_print`(expr, wrap_line=True,
num_columns=None,
use_unicode=None,
full_prec='auto', order=None,
use_unicode_sqrt_char=True)

Prints expr in pretty form.

`pprint` is just a shortcut for this function.

Parameters `expr` : expression

The expression to print.

wrap_line : bool, optional (default=True)

Line wrapping enabled/disabled.

num_columns : int or None, optional (default=None)

Number of columns before line breaking (default to None which reads the terminal width), useful when using SymPy without terminal.

use_unicode : bool or None, optional (default=None)

Use unicode characters, such as the Greek letter pi instead of the string pi.

full_prec : bool or string, optional (default="auto")

Use full precision.

order : bool or string, optional (default=None)

Set to 'none' for long expressions if slow; default is None.

use_unicode_sqrt_char : bool, optional (default=True)

Use compact single-character square root symbol (when unambiguous).

5.19.3 C code printers

This class implements C code printing, i.e. it converts Python expressions to strings of C code (see also C89CodePrinter).

Usage:

```
>>> from sympy.printing import print_ccode
>>> from sympy.functions import sin, cos, Abs, gamma
>>> from sympy.abc import x
>>> print_ccode(sin(x)**2 + cos(x)**2, standard='C89')
pow(sin(x), 2) + pow(cos(x), 2)
>>> print_ccode(2*x + cos(x), assign_to="result", standard='C89')
result = 2*x + cos(x);
>>> print_ccode(Abs(x**2), standard='C89')
fabs(pow(x, 2))
>>> print_ccode(gamma(x**2), standard='C99')
tgamma(pow(x, 2))
```

```
sympy.printing.ccode.known_functions_C89 = {'Abs': [(<function <lambda> at 0x7f0a6dc11510>
```

```
sympy.printing.ccode.known_functions_C99 = {'Abs': [(<function <lambda> at 0x7f0a6dc11510>
```

```
class sympy.printing.ccode.C89CodePrinter(settings={})
```

A printer to convert python expressions to strings of c code

Attributes

math_macros	

```
printmethod = '_ccode'
```

```
indent_code(code)
```

Accepts a string of code or a list of code lines

```
class sympy.printing.ccode.C99CodePrinter(settings={})
```

Attributes

math_macros	

```
printmethod = '_ccode'
```

```
sympy.printing.ccode.ccode(expr, assign_to=None, standard='c99', **settings)
```

Converts an expr to a string of c code

Parameters **expr** : Expr

A sympy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

standard : str, optional

String specifying the standard. If your compiler supports a more modern standard you may set this to 'c99' to allow the printer to use more math functions. [default='c89'].

precision : integer, optional

The precision for numbers such as pi [default=17].

user_functions : dict, optional

A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)] or [(argument_test, cfunction_formatter)]. See below for examples.

dereference : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if dereference=[a], the resulting code would print (*a) instead of a.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import ccode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> expr = (2*tau)**Rational(7, 2)
>>> ccode(expr)
'8*M_SQRT2*pow(tau, 7.0/2.0)'
>>> ccode(expr, math_macros={})
'8*sqrt(2)*pow(tau, 7.0/2.0)'
>>> ccode(sin(x), assign_to="s")
's = sin(x);'
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, type_aliases={real: float80})
'8*M_SQRT2l*powl(tau, 7.0L/2.0L)'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "Abs": [(lambda x: not x.is_integer, "fabs"),
...             (lambda x: x.is_integer, "ABS")],
...     "func": "f"
... }
>>> func = Function('func')
>>> ccode(func(Abs(x) + ceiling(x)), standard='C89', user_functions=custom_
↪ functions)
'f(fabs(x) + CEIL(x))'
```

or if the C-function takes a subset of the original arguments:

```
>>> ccode(2**x + 3**x, standard='C99', user_functions={'Pow': [
...     (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
...     (lambda b, e: b != 2, 'pow')])})
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, tau, standard='C89'))
if (x > 0) {
tau = x + 1;
}
else {
tau = x;
}
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False, standard='C89')
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
```

(continues on next page)

(continued from previous page)

```
>>> A = MatrixSymbol('A', 3, 1)
>>> print(ccode(mat, A, standard='C89'))
A[0] = pow(x, 2);
if (x > 0) {
    A[1] = x + 1;
}
else {
    A[1] = x;
}
A[2] = sin(x);
```

`sympy.printing.ccode.print_ccode(expr, **settings)`
 Prints C representation of the given expression.

5.19.4 C++ code printers

This module contains printers for C++ code, i.e. functions to convert SymPy expressions to strings of C++ code.

Usage:

```
>>> from sympy.printing.cxxcode import cxxcode
>>> from sympy.functions import Min, gamma
>>> from sympy.abc import x
>>> print(cxxcode(Min(gamma(x) - 1, x), standard='C++11'))
std::min(x, std::tgamma(x) - 1)
```

`class sympy.printing.cxxcode.CXX98CodePrinter(settings=None)`

Attributes

math_macros	

`printmethod = '_cxxcode'`

`class sympy.printing.cxxcode.CXX11CodePrinter(settings=None)`

Attributes

math_macros	

`printmethod = '_cxxcode'`

`sympy.printing.cxxcode.cxxcode(expr, assign_to=None, standard='c++11', **settings)`

C++ equivalent of `sympy.ccode()`.

5.19.5 RCodePrinter

This class implements R code printing (i.e. it converts Python expressions to strings of R code).

Usage:

```
>>> from sympy.printing import print_rcode
>>> from sympy.functions import sin, cos, Abs
>>> from sympy.abc import x
>>> print_rcode(sin(x)**2 + cos(x)**2)
sin(x)^2 + cos(x)^2
>>> print_rcode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_rcode(Abs(x**2))
abs(x^2)
```

```
sympy.printing.rcode.known_functions = {'Abs': 'abs', 'Max': 'max', 'Min': 'min', 'acos':
```

```
class sympy.printing.rcode.RCodePrinter(settings={})
```

A printer to convert python expressions to strings of R code

printmethod = `'_rcode'`

indent_code(code)

Accepts a string of code or a list of code lines

```
sympy.printing.rcode.rcode(expr, assign_to=None, **settings)
```

Converts an expr to a string of r code

Parameters **expr** : Expr

A sympy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=15].

user_functions : dict, optional

A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired R string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, rfunction_string)] or [(argument_test, rfunction_formatter)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the

user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import rcode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> rcode((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau^(7.0/2.0)'
>>> rcode(sin(x), assign_to="s")
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "Abs": [(lambda x: not x.is_integer, "fabs"),
...             (lambda x: x.is_integer, "ABS")],
...     "func": "f"
... }
>>> func = Function('func')
>>> rcode(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the R-function takes a subset of the original arguments:

```
>>> rcode(2**x + 3**x, user_functions={'Pow': [
...     (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
...     (lambda b, e: b != 2, 'pow')]])
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(rcode(expr, assign_to=tau))
tau = ifelse(x > 0,x + 1,x);
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> rcode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a `Matrix`:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rcode(mat, A))
A[0] = x^2;
A[1] = ifelse(x > 0,x + 1,x);
A[2] = sin(x);
```

`sympy.printing.rcode.print_rcode(expr, **settings)`
Prints R representation of the given expression.

5.19.6 Fortran Printing

The `fcode` function translates a sympy expression into Fortran code. The main purpose is to take away the burden of manually translating long mathematical expressions. Therefore the resulting expression should also require no (or very little) manual tweaking to make it compilable. The optional arguments of `fcode` can be used to fine-tune the behavior of `fcode` in such a way that manual changes in the result are no longer needed.

`sympy.printing.fcode.fcode(expr, assign_to=None, **settings)`
Converts an `expr` to a string of fortran code

Parameters `expr` : Expr

A sympy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

DEPRECATED. Use `type_mappings` instead. The precision for numbers such as `pi` [default=17].

user_functions : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If `True`, the result is a single string that may contain some constant declarations for the number symbols. If `False`, the same information is returned in a tuple of (`symbols_to_declare`, `not_supported_functions`, `code_text`). [default=`True`].

contract: bool, optional

If `True`, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the

user is responsible to provide values for the indices in the code. [default=True].

source_format : optional

The source format can be either 'fixed' or 'free'. [default='fixed']

standard : integer, optional

The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

name_mangling : bool, optional

If True, then the variables that would become identical in case-insensitive Fortran are mangled by appending different number of _ at the end. If False, SymPy won't interfere with naming of variables. [default=True]

Examples

```
>>> from sympy import fcode, symbols, Rational, sin, ceiling, floor
>>> x, tau = symbols("x, tau")
>>> fcode((2*tau)**Rational(7, 2))
'      8*sqrt(2.0d0)*tau**(7.0d0/2.0d0)'
>>> fcode(sin(x), assign_to="s")
's = sin(x)'
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "floor": [(lambda x: not x.is_integer, "FLOOR1"),
...               (lambda x: x.is_integer, "FLOOR2")]
... }
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
'      CEIL(x) + FLOOR1(x)'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
      if (x > 0) then
        tau = x + 1
      else
        tau = x
      end if
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
'      Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i))'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
      A(1, 1) = x**2
          if (x > 0) then
      A(2, 1) = x + 1
          else
      A(2, 1) = x
          end if
      A(3, 1) = sin(x)
```

`sympy.printing.fcode.print_fcode(expr, **settings)`

Prints the Fortran representation of the given expression.

See `fcode` for the meaning of the optional arguments.

class `sympy.printing.fcode.FCodePrinter(settings={})`

A printer to convert sympy expressions to strings of Fortran code

printmethod = `'_fcode'`

indent_code(code)

Accepts a string of code or a list of code lines

Two basic examples:

```
>>> from sympy import *
>>> x = symbols("x")
>>> fcode(sqrt(1-x**2))
'      sqrt(-x**2 + 1)'
>>> fcode((3 + 4*I)/(1 - conjugate(x)))
'      (cmplx(3,4))/(-conjg(x) + 1)'
```

An example where line wrapping is required:

```
>>> expr = sqrt(1-x**2).series(x,n=20).removeO()
>>> print(fcode(expr))
      -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
      @ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
      @ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
      @ /2.0d0*x**2 + 1
```

In case of line wrapping, it is handy to include the assignment so that lines are wrapped properly when the assignment part is added.

```
>>> print(fcode(expr, assign_to="var"))
var = -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
@ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
@ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
@ /2.0d0*x**2 + 1
```

For piecewise functions, the `assign_to` option is mandatory:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), assign_to="var"))
if (x < 1) then
  var = x
else
  var = x**2
end if
```

Note that by default only top-level piecewise functions are supported due to the lack of a conditional operator in Fortran 77. Inline conditionals can be supported using the `merge` function introduced in Fortran 95 by setting of the `kwargs` `standard=95`:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), standard=95))
merge(x, x**2, x < 1)
```

Loops are generated if there are Indexed objects in the expression. This also requires use of the `assign_to` option.

```
>>> A, B = map(IndexedBase, ['A', 'B'])
>>> m = Symbol('m', integer=True)
>>> i = Idx('i', m)
>>> print(fcode(2*B[i], assign_to=A[i]))
do i = 1, m
  A(i) = 2*B(i)
end do
```

Repeated indices in an expression with Indexed objects are interpreted as summation. For instance, code for the trace of a matrix can be generated with

```
>>> print(fcode(A[i, i], assign_to=x))
x = 0
do i = 1, m
  x = x + A(i, i)
end do
```

By default, number symbols such as `pi` and `E` are detected and defined as Fortran parameters. The precision of the constants can be tuned with the `precision` argument. Parameter definitions are easily avoided using the `N` function.

```
>>> print(fcode(x - pi**2 - E))
parameter (E = 2.7182818284590452d0)
parameter (pi = 3.1415926535897932d0)
x - pi**2 - E
>>> print(fcode(x - pi**2 - E, precision=25))
parameter (E = 2.718281828459045235360287d0)
parameter (pi = 3.141592653589793238462643d0)
x - pi**2 - E
```

(continues on next page)

(continued from previous page)

```
>>> print(fcode(N(x - pi**2, 25)))
x - 9.869604401089358618834491d0
```

When some functions are not part of the Fortran standard, it might be desirable to introduce the names of user-defined functions in the Fortran expression.

```
>>> print(fcode(1 - gamma(x)**2, user_functions={'gamma': 'mygamma'}))
-mygamma(x)**2 + 1
```

However, when the `user_functions` argument is not provided, `fcode` attempts to use a reasonable default and adds a comment to inform the user of the issue.

```
>>> print(fcode(1 - gamma(x)**2))
C      Not supported in Fortran:
C      gamma
      -gamma(x)**2 + 1
```

By default the output is human readable code, ready for copy and paste. With the option `human=False`, the return value is suitable for post-processing with source code generators that write routines with multiple instructions. The return value is a three-tuple containing: (i) a set of number symbols that must be defined as 'Fortran parameters', (ii) a list functions that cannot be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```
>>> fcode(1 - gamma(x)**2, human=False)
(set(), {gamma(x)}, '      -gamma(x)**2 + 1')
>>> fcode(1 - sin(x)**2, human=False)
(set(), set(), '      -sin(x)**2 + 1')
>>> fcode(x - pi**2, human=False)
(({pi, '3.1415926535897932d0'}), set(), '      x - pi**2')
```

5.19.7 Mathematica code printing

```
sympy.printing.mathematica.known_functions = {'acos': [(<function <lambda> at 0x7f0a6db55f
```

```
class sympy.printing.mathematica.MCodePrinter(settings={})
```

A printer to convert python expressions to strings of the Wolfram's Mathematica code

`printmethod = '_mcode'`

`doprint(expr)`

Returns printer's representation for `expr` (as a string)

```
sympy.printing.mathematica.mathematica_code(expr, **settings)
```

Converts an `expr` to a string of the Wolfram Mathematica code

Examples

```
>>> from sympy import mathematica_code as mcode, symbols, sin
>>> x = symbols('x')
>>> mcode(sin(x).series(x).removeO())
'(1/120)*x^5 - 1/6*x^3 + x'
```


5.19.8 Javascript Code printing

```
sympy.printing.jcode.known_functions = {'Abs': 'Math.abs', 'Max': 'Math.max', 'Min': 'Mat
```

```
class sympy.printing.jcode.JavascriptCodePrinter(settings={})
```

“A Printer to convert python expressions to strings of javascript code

```
printmethod = '_javascript'
```

```
indent_code(code)
```

Accepts a string of code or a list of code lines

```
sympy.printing.jcode.jcode(expr, assign_to=None, **settings)
```

Converts an expr to a string of javascript code

Parameters `expr` : Expr

A sympy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=15].

user_functions : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, js_function_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import jcode, symbols, Rational, sin, ceiling, Abs
>>> x, tau = symbols("x, tau")
>>> jcode((2*tau)**Rational(7, 2))
'8*Math.sqrt(2)*Math.pow(tau, 7/2)'
>>> jcode(sin(x), assign_to="s")
's = Math.sin(x);'
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, js_function_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "Abs": [(lambda x: not x.is_integer, "fabs"),
...             (lambda x: x.is_integer, "ABS")]
... }
>>> jscode(Abs(x) + ceiling(x), user_functions=custom_functions)
'fabs(x) + CEIL(x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(jscode(expr, tau))
if (x > 0) {
    tau = x + 1;
}
else {
    tau = x;
}
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> jscode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(jscode(mat, A))
A[0] = Math.pow(x, 2);
if (x > 0) {
    A[1] = x + 1;
}
else {
    A[1] = x;
}
A[2] = Math.sin(x);
```

If True, we try to create single-statement code instead of multiple statements. [default=True].

Examples

```
>>> from sympy import julia_code, symbols, sin, pi
>>> x = symbols('x')
>>> julia_code(sin(x).series(x).removeO())
'x.^5/120 - x.^3/6 + x'
```

```
>>> from sympy import Rational, ceiling, Abs
>>> x, y, tau = symbols("x, y, tau")
>>> julia_code((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau.^(7/2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its possible in Julia to write “vectorized” code. It is harmless if the values are scalars.

```
>>> julia_code(sin(pi*x*y), assign_to="s")
's = sin(pi*x.*y)'
```

If you need a matrix product “*” or matrix power “^”, you can specify the symbol as a MatrixSymbol.

```
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> julia_code(3*pi*A**3)
'(3*pi)*A^3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use “*”, Symbols use “.” and MatrixSymbols use “*”. A HadamardProduct can be used to specify componentwise multiplication “.*” of two MatrixSymbols. There is currently there is no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write “(x^2*y)*A^3”, we generate:

```
>>> julia_code(x**2*y*A**3)
'(x.^2.*y)*A^3'
```

Matrices are supported using Julia inline notation. When using assign_to with matrices, the name can be specified either as a string or as a MatrixSymbol. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]])
>>> julia_code(mat, assign_to='A')
'A = [x.^2 sin(x) ceil(x)]'
```

Piecewise expressions are implemented with logical masking by default. Alternatively, you can pass “inline=False” to use if-else conditionals. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> pw = Piecewise((x + 1, x > 0), (x, True))
>>> julia_code(pw, assign_to=tau)
'tau = ((x > 0) ? (x + 1) : (x))'
```

```
>>> mat = Matrix([[x**2, pw, sin(x)]])
>>> julia_code(mat, assign_to='A')
'A = [x.^2 ((x > 0) ? (x + 1) : (x)) sin(x)]'
```

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
...     "f": "existing_julia_fcn",
...     "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...            (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> julia_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_julia_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])'
```

```
>>> from sympy import Eq, IndexedBase, Idx, ccode
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> julia_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])./(t[i + 1] - t[i])'
```

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This can be helpful for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=16].

user_functions : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

inline: bool, optional

If True, we try to create single-statement code instead of multiple statements. [default=True].

Examples

```
>>> from sympy import octave_code, symbols, sin, pi
>>> x = symbols('x')
>>> octave_code(sin(x).series(x).remove0())
'x.^5/120 - x.^3/6 + x'
```

```
>>> from sympy import Rational, ceiling, Abs
>>> x, y, tau = symbols("x, y, tau")
>>> octave_code((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau.^(7/2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its very common in Octave to write “vectorized” code. It is harmless if the values are scalars.

```
>>> octave_code(sin(pi*x*y), assign_to="s")
's = sin(pi*x.*y);'
```

If you need a matrix product “*” or matrix power “^”, you can specify the symbol as a MatrixSymbol.

```
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> octave_code(3*pi*A**3)
'(3*pi)*A^3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use “*”, Symbols use “.” and MatrixSymbols use “*”. A HadamardProduct can be used to specify componentwise multiplication “.” of two MatrixSymbols. There is currently no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write “ $(x^2y)A^3$ ”, we generate:

```
>>> octave_code(x**2*y*A**3)
'(x.^2.*y)*A^3'
```

Matrices are supported using Octave inline notation. When using `assign_to` with matrices, the name can be specified either as a string or as a `MatrixSymbol`. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]])
>>> octave_code(mat, assign_to='A')
'A = [x.^2 sin(x) ceil(x)];'
```

Piecewise expressions are implemented with logical masking by default. Alternatively, you can pass “`inline=False`” to use if-else conditionals. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> pw = Piecewise((x + 1, x > 0), (x, True))
>>> octave_code(pw, assign_to=tau)
'tau = ((x > 0).*(x + 1) + ~(x > 0)).*(x);'
```

Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([[x**2, pw, sin(x)]])
>>> octave_code(mat, assign_to='A')
'A = [x.^2 ((x > 0).*(x + 1) + ~(x > 0)).*(x)) sin(x)];'
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e., `[(argument_test, cfunction_string)]`. This can be used to call a custom Octave function.

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
...     "f": "existing_octave_fcn",
...     "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...            (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])'
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx, ccode
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> octave_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy(i) = (y(i + 1) - y(i))./(t(i + 1) - t(i));'
```

5.19.11 Rust code printing

`sympy.printing.rust.known_functions = {'': 'ln_lp', 'Abs': 'abs', 'Max': 'max', 'Min': 'mi`

class `sympy.printing.rust.RustCodePrinter`(settings={})

A printer to convert python expressions to strings of Rust code

printmethod = `'_rust_code'`

indent_code(code)

Accepts a string of code or a list of code lines

`sympy.printing.rust.rust_code`(expr, assign_to=None, **settings)

Converts an expr to a string of Rust code

Parameters `expr` : Expr

A sympy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=15].

user_functions : dict, optional

A dictionary where the keys are string representations of either `FunctionClass` or `UndefinedFunction` instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

dereference : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import rust_code, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> rust_code((2*tau)**Rational(7, 2))
'8*1.4142135623731*tau.powf(7_f64/2.0)'
>>> rust_code(sin(x), assign_to="s")
's = x.sin();'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "Abs": [(lambda x: not x.is_integer, "fabs", 4),
...             (lambda x: x.is_integer, "ABS", 4)],
...     "func": "f"
... }
>>> func = Function('func')
>>> rust_code(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'(fabs(x) + x.CEIL()).f()'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(rust_code(expr, tau))
tau = if (x > 0) {
    x + 1
} else {
    x
};
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
```

(continues on next page)

(continued from previous page)

```
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> rust_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rust_code(mat, A))
A = [x.powi(2), if (x > 0) {
      x + 1
    } else {
      x
    }, x.sin()];
```

5.19.12 Theano Code printing

class `sympy.printing.theanocode.TheanoPrinter(*args, **kwargs)`

Code printer for Theano computations

printmethod = `'_theano'`

doprint(`expr, **kwargs`)

Returns printer's representation for `expr` (as a string)

`sympy.printing.theanocode.theano_function`(`inputs, outputs, dtypes={}, cache=None, **kwargs`)

Create Theano function from SymPy expressions

5.19.13 Gtk

You can print to a `gtkmathview` widget using the function `print_gtk` located in `sympy.printing.gtk` (it requires to have installed `gtkmathview` and `libgtkmathview-bin` in some systems).

`GtkMathView` accepts MathML, so this rendering depends on the MathML representation of the expression.

Usage:

```
from sympy import *
print_gtk(x**2 + 2*exp(x**3))
```

`sympy.printing.gtk.print_gtk`(`x, start_viewer=True`)

Print to `Gtkmathview`, a `gtk` widget capable of rendering MathML.

Needs `libgtkmathview-bin`


```
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
```

```
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{7 / 2}$
```

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
```

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
```

itex: Specifies if itex-specific syntax is used, including emitting $\$$ $\$$.

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))
$$8 \sqrt{2} \mu^{\frac{7}{2}}$$
```

fold_frac_powers: Emit “ p/q ” instead of “ $\frac{p}{q}$ ” for fractional powers.

```
>>> print(latex((2*tau)**Rational(7,2), fold_frac_powers=True))
8 \sqrt{2} \tau^{7/2}
```

fold_func_brackets: Fold function brackets where applicable.

```
>>> print(latex((2*tau)**sin(Rational(7,2))))
\left(2 \tau\right)^{\sin\left(\frac{7}{2}\right)}
>>> print(latex((2*tau)**sin(Rational(7,2)), fold_func_brackets = True))
\left(2 \tau\right)^{\sin \frac{7}{2}}
```

fold_short_frac: Emit “ p/q ” instead of “ $\frac{p}{q}$ ” when the denominator is simple enough (at most two terms and no powers). The default value is *True* for inline mode, *False* otherwise.

```
>>> print(latex(3*x**2/y))
\frac{3 x^2}{y}
>>> print(latex(3*x**2/y, fold_short_frac=True))
3 x^2 / y
```

long_frac_ratio: The allowed ratio of the width of the numerator to the width of the denominator before we start breaking off long fractions. If *None* (the default value), long fractions are not broken up.

```
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=2))
\frac{\int r\, dr}{2 \pi}
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=0))
\frac{1}{2 \pi} \int r\, dr
```

mul_symbol: The symbol to use for multiplication. Can be one of *None*, “*ldot*”, “*dot*”, or “*times*”.

```
>>> print(latex((2*tau)**sin(Rational(7,2)), mul_symbol="times"))
\left(2 \times \tau\right)^{\sin\left(\frac{7}{2}\right)}
```

inv_trig_style: How inverse trig functions should be displayed. Can be one of “*abbreviated*”, “*full*”, or “*power*”. Defaults to “*abbreviated*”.

```
>>> print(latex(asin(Rational(7,2))))
\operatorname{asin}{\left (\frac{7}{2} \right )}
>>> print(latex(asin(Rational(7,2)), inv_trig_style="full"))
\arcsin{\left (\frac{7}{2} \right )}
>>> print(latex(asin(Rational(7,2)), inv_trig_style="power"))
\sin^{-1}{\left (\frac{7}{2} \right )}
```

mat_str: Which matrix environment string to emit. “smallmatrix”, “matrix”, “array”, etc. Defaults to “smallmatrix” for inline mode, “matrix” for matrices of no more than 10 columns, and “array” otherwise.

```
>>> print(latex(Matrix(2, 1, [x, y])))
\left[\begin{matrix}x\\y\end{matrix}\right]
```

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_str = "array"))
\left[\begin{array}{c}x\\y\end{array}\right]
```

mat_delim: The delimiter to wrap around matrices. Can be one of “[”, “(”, or the empty string. Defaults to “[”.

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_delim="("))
\left(\begin{matrix}x\\y\end{matrix}\right)
```

symbol_names: Dictionary of symbols and the custom strings they should be emitted as.

```
>>> print(latex(x**2, symbol_names={x: 'x_i'}))
x_i^2
```

latex also supports the builtin container types list, tuple, and dictionary.

```
>>> print(latex([2/x, y], mode='inline'))
$\left [ 2 / x, \quad y \right ]$
```

ln_notation: If set to True “ln” is used instead of default “log”

```
>>> print(latex(log(10)))
\log{\left (10 \right )}
```

```
>>> print(latex(log(10), ln_notation=True))
\ln{\left (10 \right )}
```

sympy.printing.latex.**print_latex**(expr, **settings)
Prints LaTeX representation of the given expression.

5.19.16 MathMLPrinter

This class is responsible for MathML printing. See `sympy.printing.mathml`.

More info on mathml : <http://www.w3.org/TR/MathML2>

class sympy.printing.mathml.**MathMLPrinterBase**(settings=None)
Contains common code required for MathMLContentPrinter and MathMLPresentationPrinter.

Attributes

printmethod	

class sympy.printing.mathml.MathMLContentPrinter(settings=None)

Prints an expression to the Content MathML markup language.

References: <https://www.w3.org/TR/MathML2/chapter4.html>

printmethod = `'_mathml_content'`

mathml_tag(e)

Returns the MathML tag for an expression.

class sympy.printing.mathml.MathMLPresentationPrinter(settings=None)

Prints an expression to the Presentation MathML markup language.

References: <https://www.w3.org/TR/MathML2/chapter3.html>

printmethod = `'_mathml_presentation'`

mathml_tag(e)

Returns the MathML tag for an expression.

sympy.printing.mathml.mathml(expr, printer='content', **settings)

Returns the MathML representation of expr. If printer is presentation then prints Presentation MathML else prints content MathML.

sympy.printing.mathml.print_mathml(expr, printer='content', **settings)

Prints a pretty representation of the MathML code for expr. If printer is presentation then prints Presentation MathML else prints content MathML.

Examples

```
>>> ##
>>> from sympy.printing.mathml import print_mathml
>>> from sympy.abc import x
>>> print_mathml(x+1)
<apply>
  <plus/>
  <ci>x</ci>
  <cn>1</cn>
</apply>
>>> print_mathml(x+1, printer='presentation')
<mrow>
  <mi>x</mi>
  <mo>+</mo>
  <mn>1</mn>
</mrow>
```

5.19.17 PythonCodePrinter

Python code printers

This module contains python code printers for plain python as well as NumPy & SciPy enabled code.

class `sympy.printing.pycode.MpmathPrinter`(settings=None)
 Lambda printer for mpmath which maintains precision for floats

Attributes

modules	

class `sympy.printing.pycode.NumPyPrinter`(settings=None)
 Numpy printer which handles vectorized piecewise functions, logical operators, etc.

Attributes

modules	

`sympy.printing.pycode.pycode`(expr, **settings)
 Converts an expr to a string of Python code

Parameters `expr` : Expr

A SymPy expression.

fully_qualified_modules : bool

Whether or not to write out full module names of functions (`math.sin` vs. `sin`). default: True.

Examples

```
>>> from sympy import tan, Symbol
>>> from sympy.printing.pycode import pycode
>>> pycode(tan(Symbol('x')) + 1)
'math.tan(x) + 1'
```

5.19.18 PythonPrinter

This class implements Python printing. Usage:

```
>>> from sympy import print_python, sin
>>> from sympy.abc import x

>>> print_python(5*x**3 + sin(x))
x = Symbol('x')
e = 5*x**3 + sin(x)
```

5.19.19 srepr

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

`srepr()` gives more low level textual output than `repr()`

Example:

```
>>> repr(5*x**3 + sin(x))
'5*x**3 + sin(x)'

>>> srepr(5*x**3 + sin(x))
"Add(Mul(Integer(5), Pow(Symbol('x'), Integer(3))), sin(Symbol('x')))"
```

`srepr()` gives the `repr` form, which is what `repr()` would normally give but for SymPy we don't actually use `srepr()` for `__repr__` because it's so verbose, it is unlikely that anyone would want it called by default. Another reason is that lists call `repr` on their elements, like `print([a, b, c])` calls `repr(a)`, `repr(b)`, `repr(c)`. So if we used `srepr` for `__repr__` any list with SymPy objects would include the `srepr` form, even if we used `str()` or `print()`.

class `sympy.printing.repr.ReprPrinter`(settings=None)

printmethod = `'_sympyrepr'`

emptyPrinter(expr)

The fallback printer.

reprify(args, sep)

Prints each item in *args* and joins them with *sep*.

`sympy.printing.repr.srepr`(expr, **settings)

return expr in repr form

5.19.20 StrPrinter

This module generates readable representations of SymPy expressions.

class `sympy.printing.str.StrPrinter`(settings=None)

printmethod = `'_sympystr'`

`sympy.printing.str.sstrrepr`(expr, **settings)

return expr in mixed str/repr form

i.e. strings are returned in repr form with quotes, and everything else is returned in str form.

This function could be useful for hooking into `sys.displayhook`

5.19.21 Tree Printing

The functions in this module create a representation of an expression as a tree.

`sympy.printing.tree.pprint_nodes`(subtrees)

Prettyprints systems of nodes.

Examples


```
>>> from sympy.printing.tree import pprint_nodes
>>> print(pprint_nodes(["a", "b1\nb2", "c"]))
+-a
+-b1
| b2
+-c
```

`sympy.printing.tree.print_node(node)`
Returns information about the “node”.

This includes class name, string representation and assumptions.

`sympy.printing.tree.tree(node)`
Returns a tree representation of “node” as a string.

It uses `print_node()` together with `pprint_nodes()` on `node.args` recursively.

See also: `print_tree()`

`sympy.printing.tree.print_tree(node)`
Prints a tree representation of “node”.

Examples

```
>>> from sympy.printing import print_tree
>>> from sympy import Symbol
>>> x = Symbol('x', odd=True)
>>> y = Symbol('y', even=True)
>>> print_tree(y**x)
Pow: y**x
+-Symbol: y
| algebraic: True
| commutative: True
| complex: True
| even: True
| hermitian: True
| imaginary: False
| integer: True
| irrational: False
| noninteger: False
| odd: False
| rational: True
| real: True
| transcendental: False
+-Symbol: x
| algebraic: True
| commutative: True
| complex: True
| even: False
| hermitian: True
| imaginary: False
| integer: True
| irrational: False
| noninteger: False
| nonzero: True
| odd: True
| rational: True
```

(continues on next page)

(continued from previous page)

```
real: True
transcendental: False
zero: False
```

See also: `tree()`

5.19.22 Preview

A useful function is `preview`:

```
sympy.printing.preview.preview(expr, output='png', viewer=None, euler=True,
                                packages=(), filename=None, outputbuffer=None,
                                preamble=None, dvioptions=None, outputTex-
                                File=None, **latex_settings)
```

View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.

If the `expr` argument is an expression, it will be exported to LaTeX and then compiled using the available TeX distribution. The first argument, `'expr'`, may also be a LaTeX string. The function will then run the appropriate viewer for the given output format or use the user defined one. By default png output is generated.

By default pretty Euler fonts are used for typesetting (they were used to typeset the well known “Concrete Mathematics” book). For that to work, you need the `'eulervm.sty'` LaTeX style (in Debian/Ubuntu, install the `texlive-fonts-extra` package). If you prefer default AMS fonts or your system lacks `'eulervm'` LaTeX package then unset the `'euler'` keyword argument.

To use viewer auto-detection, lets say for `'png'` output, issue

```
>>> from sympy import symbols, preview, Symbol
>>> x, y = symbols("x,y")
```

```
>>> preview(x + y, output='png')
```

This will choose `'pyglet'` by default. To select a different one, do

```
>>> preview(x + y, output='png', viewer='gimp')
```

The `'png'` format is considered special. For all other formats the rules are slightly different. As an example we will take `'dvi'` output format. If you would run

```
>>> preview(x + y, output='dvi')
```

then `'view'` will look for available `'dvi'` viewers on your system (predefined in the function, so it will try `evince`, first, then `kdvi` and `xdvi`). If nothing is found you will need to set the viewer explicitly.

```
>>> preview(x + y, output='dvi', viewer='superior-dvi-viewer')
```

This will skip auto-detection and will run user specified `'superior-dvi-viewer'`. If `'view'` fails to find it on your system it will gracefully raise an exception.

You may also enter `'file'` for the viewer argument. Doing so will cause this function to return a file object in read-only mode, if `'filename'` is unset. However, if it was set, then `'preview'` writes the generated file to this filename instead.

There is also support for writing to a BytesIO like object, which needs to be passed to the 'outputbuffer' argument.

```
>>> from io import BytesIO
>>> obj = BytesIO()
>>> preview(x + y, output='png', viewer='BytesIO',
...         outputbuffer=obj)
```

The LaTeX preamble can be customized by setting the 'preamble' keyword argument. This can be used, e.g., to set a different font size, use a custom documentclass or import certain set of LaTeX packages.

```
>>> preamble = "\\documentclass[10pt]{article}\\n" \
...           "\\usepackage{amsmath,amsfonts}\\begin{document}"
>>> preview(x + y, output='png', preamble=preamble)
```

If the value of 'output' is different from 'dvi' then command line options can be set ('dviop-tions' argument) for the execution of the 'dvi'+output conversion tool. These options have to be in the form of a list of strings (see subprocess.Popen).

Additional keyword args will be passed to the latex call, e.g., the symbol_names flag.

```
>>> phidd = Symbol('phidd')
>>> preview(phidd, symbol_names={phidd:r'\ddot{\varphi}'})
```

For post-processing the generated TeX File can be written to a file by passing the desired filename to the 'outputTexFile' keyword argument. To write the TeX code to a file named "sample.tex" and run the default png viewer to display the resulting bitmap, do

```
>>> preview(x + y, outputTexFile="sample.tex")
```

5.19.23 Implementation - Helper Classes/Functions

`sympy.printing.conventions.split_super_sub(text)`

Split a symbol name into a name, superscripts and subscripts

The first part of the symbol name is considered to be its actual 'name', followed by super- and subscripts. Each superscript is preceded with a "^" character or by "_". Each subscript is preceded by a "_" character. The three return values are the actual name, a list with superscripts and a list with subscripts.

```
>>> from sympy.printing.conventions import split_super_sub
>>> split_super_sub('a_x^1')
('a', ['1'], ['x'])
>>> split_super_sub('var_sub1__sup_sub2')
('var', ['sup'], ['sub1', 'sub2'])
```

CodePrinter

This class is a base class for other classes that implement code-printing functionality, and additionally lists a number of functions that cannot be easily translated to C or Fortran.

class `sympy.printing.codeprinter.Assignment`

Represents variable assignment for code generation.

Parameters `lhs` : Expr

Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include `Symbol`, `MatrixSymbol`, `MatrixElement`, and `Indexed`. Types that subclass these types are also supported.

rhs : Expr

Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a `Matrix` type can be assigned to `MatrixSymbol`, but not to `Symbol`, as the dimensions will not align.

Examples

```
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> from sympy.codegen.ast import Assignment
>>> x, y, z = symbols('x, y, z')
>>> Assignment(x, y)
Assignment(x, y)
>>> Assignment(x, 0)
Assignment(x, 0)
>>> A = MatrixSymbol('A', 1, 3)
>>> mat = Matrix([x, y, z]).T
>>> Assignment(A, mat)
Assignment(A, Matrix([[x, y, z]]))
>>> Assignment(A[0, 1], x)
Assignment(A[0, 1], x)
```

class `sympy.printing.codeprinter.CodePrinter`(settings=None)

The base class for code-printing subclasses.

printmethod = `'_sympystr'`

exception `sympy.printing.codeprinter.AssignmentError`

Raised if an assignment variable for a loop is missing.

Precedence

`sympy.printing.precedence.PRECEDENCE` = {'Add': 40, 'And': 30, 'Atom': 1000, 'BitwiseAnd': ...
Default precedence values for some basic types.

`sympy.printing.precedence.PRECEDENCE_VALUES` = {'Add': 40, 'And': 30, 'Equality': 50, 'Equi...
A dictionary assigning precedence values to certain classes. These values are treated like they were inherited, so not every single class has to be named here.

`sympy.printing.precedence.PRECEDENCE_FUNCTIONS` = {'Float': <function precedence_Float at 0...
Sometimes it's not enough to assign a fixed precedence value to a class. Then a function can be inserted in this dictionary that takes an instance of this class as argument and returns the appropriate precedence value.

`sympy.printing.precedence.precedence`(item)
Returns the precedence of a given object.

5.19.24 Pretty-Printing Implementation Helpers

`sympy.printing.pretty.pretty_symbology.U(name)`
 unicode character by name or None if not found

`sympy.printing.pretty.pretty_symbology.pretty_use_unicode(flag=None)`
 Set whether pretty-printer should use unicode by default

`sympy.printing.pretty.pretty_symbology.pretty_try_use_unicode()`
 See if unicode output is available and leverage it if possible

`sympy.printing.pretty.pretty_symbology.xstr(*args)`
 call str or unicode depending on current mode

The following two functions return the Unicode version of the inputted Greek letter.

`sympy.printing.pretty.pretty_symbology.g(l)`

`sympy.printing.pretty.pretty_symbology.G(l)`

`sympy.printing.pretty.pretty_symbology.greek_letters = ['alpha', 'beta', 'gamma', 'delta', ...]`
 list() -> new empty list list(iterable) -> new list initialized from iterable's items

`sympy.printing.pretty.pretty_symbology.digit_2txt = {'0': 'ZERO', '1': 'ONE', '2': 'TWO', ...}`

`sympy.printing.pretty.pretty_symbology.symb_2txt = {'(': 'LEFT PARENTHESIS', ')': 'RIGHT PARENTHESIS', ...}`

The following functions return the Unicode subscript/superscript version of the character.

`sympy.printing.pretty.pretty_symbology.sub = {'(' : '(', ')': ')', '+': '+', '-': '-', '0': '0', ...}`

`sympy.printing.pretty.pretty_symbology.sup = {'(' : '(', ')': ')', '+': '+', '-': '-', '0': '0', ...}`

The following functions return Unicode vertical objects.

`sympy.printing.pretty.pretty_symbology.xobj(symb, length)`
 Construct spatial object of given length.

return: [] of equal-length strings

`sympy.printing.pretty.pretty_symbology.vobj(symb, height)`
 Construct vertical object of a given height

see: xobj

`sympy.printing.pretty.pretty_symbology.hobj(symb, width)`
 Construct horizontal object of a given width

see: xobj

The following constants are for rendering roots and fractions.

`sympy.printing.pretty.pretty_symbology.root = {2: '√', 3: '∛', 4: '∜'}`

`sympy.printing.pretty.pretty_symbology.VF(txt)`

`sympy.printing.pretty.pretty_symbology.frac = {(1, 2): '½', (1, 3): '⅓', (1, 4): '¼', (1, ...}`

The following constants/functions are for rendering atoms and symbols.

`sympy.printing.pretty.pretty_symbology.xsym(sym)`
 get symbology for a 'character'

`sympy.printing.pretty.pretty_symbology.atoms_table = {'Complexes': 'ℂ', 'EmptySet': '∅', ...}`

`sympy.printing.pretty.pretty_symbology.pretty_atom(atom_name, fault=None)`
 de-

return pretty representation of an atom

`sympy.printing.pretty.pretty_symbology.pretty_symbol(symb_name)`
return pretty representation of a symbol

`sympy.printing.pretty.pretty_symbology.annotated(letter)`
Return a stylised drawing of the letter `letter`, together with information on how to put annotations (super- and subscripts to the left and to the right) on it.

See `pretty.py` functions `_print_meijerg`, `_print_hyper` on how to use this information.

Prettyprinter by Jurjen Bos. (I hate spammers: mail me at `pietjepuk314` at the reverse of `ku.oc.oohay`). All objects have a method that create a “stringPict”, that can be used in the `str` method for pretty printing.

Updates by Jason Gedge (email <my last name> at cs mun ca)

- `terminal_string()` method
- minor fixes and changes (mostly to `prettyForm`)

TODO:

- Allow left/center/right alignment options for above/below and top/center/bottom alignment options for left/right

class `sympy.printing.pretty.stringpict.stringPict(s, baseline=0)`

An ASCII picture. The pictures are represented as a list of equal length strings.

above(*args)

Put pictures above this picture. Returns string, baseline arguments for `stringPict`. Baseline is baseline of bottom picture.

below(*args)

Put pictures under this picture. Returns string, baseline arguments for `stringPict`. Baseline is baseline of top picture

Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("x+3").below(
...     stringPict.LINE, '3')[0])
x+3
---
 3
```

height()

The height of the picture in characters.

left(*args)

Put pictures (left to right) at left. Returns string, baseline arguments for `stringPict`.

leftslash()

Precede object by a slash of the proper size.

static next(*args)

Put a string of `stringPicts` next to each other. Returns string, baseline arguments for `stringPict`.

parens(left='(', right=')', ifascii_noughly=False)

Put parentheses around self. Returns string, baseline arguments for `stringPict`.

left or right can be `None` or empty string which means ‘no paren from that side’

render(*args, **kwargs)

Return the string form of self.

Unless the argument `line_break` is set to `False`, it will break the expression in a form that can be printed on the terminal without being broken up.

right(*args)

Put pictures next to this one. Returns string, baseline arguments for `stringPict`. (Multiline) strings are allowed, and are given a baseline of 0.

Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("10").right(" + ",stringPict("1\r-\r2",1))[0])
      1
10 + -
      2
```

root(n=None)

Produce a nice root symbol. Produces ugly results for big n inserts.

static stack(*args)

Put pictures on top of each other, from top to bottom. Returns string, baseline arguments for `stringPict`. The baseline is the baseline of the second picture. Everything is centered. Baseline is the baseline of the second picture. Strings are allowed. The special value `stringPict.LINE` is a row of '-' extended to the width.

terminal_width()

Return the terminal width if possible, otherwise return 0.

width()

The width of the picture in characters.

class `sympy.printing.pretty.stringpict.prettyForm`(s, baseline=0, binding=0, unicode=None)

Extension of the `stringPict` class that knows about basic math applications, optimizing double minus signs.

"Binding" is interpreted as follows:

```
ATOM this is an atom: never needs to be parenthesized
FUNC this is a function application: parenthesize if added (?)
DIV  this is a division: make wider division if divided
POW  this is a power: only parenthesize if exponent
MUL  this is a multiplication: parenthesize if powered
ADD  this is an addition: parenthesize if multiplied or powered
NEG  this is a negative number: optimize if added, parenthesize if
      multiplied or powered
OPEN this is an open object: parenthesize if added, multiplied, or
      powered (example: Piecewise)
```

static apply(function, *args)

Functions of one or more variables.

5.19.25 dotprint

```
sympy.printing.dot.dotprint(expr, styles=[(<class 'sympy.core.basic.Basic'>,
                                           {'color': 'blue', 'shape': 'ellipse'}), (<class
                                           'sympy.core.expr.Expr'>, {'color': 'black'})],
                             atom=<function <lambda>>, maxdepth=None,
                             repeat=True, labelfunc=<class 'str'>, **kwargs)
```

DOT description of a SymPy expression tree

Options are

styles: Styles for different classes. The default is:

```
[ (Basic, {'color': 'blue', 'shape': 'ellipse'}),
  (Expr, {'color': 'black'}) ]
```

atom: Function used to determine if an arg is an atom. The default is `lambda x: not isinstance(x, Basic)`. Another good choice is `lambda x: not x.args`.

maxdepth: The maximum depth. The default is `None`, meaning no limit.

repeat: Whether to different nodes for separate common subexpressions. The default is `True`. For example, for $x + x*y$ with `repeat=True`, it will have two nodes for x and with `repeat=False`, it will have one (warning: even if it appears twice in the same object, like `Pow(x, x)`, it will still only appear only once. Hence, with `repeat=False`, the number of arrows out of an object might not equal the number of args it has).

labelfunc: How to label leaf nodes. The default is `str`. Another good option is `srepr`. For example with `str`, the leaf nodes of $x + 1$ are labeled, x and 1 . With `srepr`, they are labeled `Symbol('x')` and `Integer(1)`.

Additional keyword arguments are included as styles for the graph.

Examples

```
>>> from sympy.printing.dot import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{

# Graph style
"ordering"="out"
"rankdir"="TD"

#####
# Nodes #
#####

"Add(Integer(2), Symbol(x))_" ["color"="black", "label"="Add", "shape"="ellipse",
    "x"="1"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol(x)_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#####
# Edges #
```

(continues on next page)

(continued from previous page)

```
#####
"Add(Integer(2), Symbol(x))_()" -> "Integer(2)_(0,)";
"Add(Integer(2), Symbol(x))_()" -> "Symbol(x)_(1,)";
}
```

5.20 Plotting Module

5.20.1 Introduction

The plotting module allows you to make 2-dimensional and 3-dimensional plots. Presently the plots are rendered using `matplotlib` as a backend. It is also possible to plot 2-dimensional plots using a `TextBackend` if you don't have `matplotlib`.

The plotting module has the following functions:

- `plot`: Plots 2D line plots.
- `plot_parametric`: Plots 2D parametric plots.
- `plot_implicit`: Plots 2D implicit and region plots.
- `plot3d`: Plots 3D plots of functions in two variables.
- `plot3d_parametric_line`: Plots 3D line plots, defined by a parameter.
- `plot3d_parametric_surface`: Plots 3D parametric surface plots.

The above functions are only for convenience and ease of use. It is possible to plot any plot by passing the corresponding `Series` class to `Plot` as argument.

5.20.2 Plot Class

class `sympy.plotting.plot.Plot(*args, **kwargs)`

The central class of the plotting module.

For interactive work the function `plot` is better suited.

This class permits the plotting of sympy expressions using numerous backends (`matplotlib`, `textplot`, the old `pyglet` module for sympy, Google charts api, etc).

The figure can contain an arbitrary number of plots of sympy expressions, lists of coordinates of points, etc. `Plot` has a private attribute `_series` that contains all data series to be plotted (expressions for lines or surfaces, lists of points, etc (all subclasses of `BaseSeries`)). Those data series are instances of classes not imported by `from sympy import *`.

The customization of the figure is on two levels. Global options that concern the figure as a whole (eg title, xlabel, scale, etc) and per-data series options (eg name) and aesthetics (eg. color, point shape, line type, etc.).

The difference between options and aesthetics is that an aesthetic can be a function of the coordinates (or parameters in a parametric plot). The supported values for an aesthetic are: - None (the backend uses default values) - a constant - a function of one variable (the first coordinate or parameter) - a function of two variables (the first and second coordinate or parameters) - a function of three variables (only in nonparametric

3D plots) Their implementation depends on the backend so they may not work in some backends.

If the plot is parametric and the arity of the aesthetic function permits it the aesthetic is calculated over parameters and not over coordinates. If the arity does not permit calculation over parameters the calculation is done over coordinates.

Only cartesian coordinates are supported for the moment, but you can use the parametric plots to plot in polar, spherical and cylindrical coordinates.

The arguments for the constructor `Plot` must be subclasses of `BaseSeries`.

Any global option can be specified as a keyword argument.

The global options for a figure are:

- `title` : str
- `xlabel` : str
- `ylabel` : str
- `legend` : bool
- `xscale` : {'linear', 'log'}
- `yscale` : {'linear', 'log'}
- `axis` : bool
- `axis_center` : tuple of two floats or {'center', 'auto'}
- `xlim` : tuple of two floats
- `ylim` : tuple of two floats
- `aspect_ratio` : tuple of two floats or {'auto'}
- `autoscale` : bool
- `margin` : float in [0, 1]

The per data series options and aesthetics are: There are none in the base series. See below for options for subclasses.

Some data series support additional aesthetics or options:

`ListSeries`, `LineOver1DRangeSeries`, `Parametric2DLineSeries`, `Parametric3DLineSeries` support the following:

Aesthetics:

- `line_color` : function which returns a float.

options:

- `label` : str
- `steps` : bool
- `integers_only` : bool

`SurfaceOver2DRangeSeries`, `ParametricSurfaceSeries` support the following:

aesthetics:

- `surface_color` : function which returns a float.

append(arg)

Adds an element from a plot's series to an existing plot.

See also:

[extend](#) (page 1061)

Examples

Consider two Plot objects, p1 and p2. To add the second plot's first series object to the first, use the append method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x)
>>> p2 = plot(x)
>>> p1.append(p2[0])
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
```

extend(arg)

Adds all series from another plot.

Examples

Consider two Plot objects, p1 and p2. To add the second plot to the first, use the extend method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x)
>>> p2 = plot(x)
>>> p1.extend(p2)
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
```

5.20.3 Plotting Function Reference

`sympy.plotting.plot.plot(*args, **kwargs)`

Plots a function of a single variable and returns an instance of the Plot class (also, see the description of the show keyword argument below).

The plotting uses an adaptive algorithm which samples recursively to accurately plot the plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

See also:

[Plot](#) (page 1059), [LineOver1DRangeSeries](#).

Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
```

Single Plot

```
>>> plot(x**2, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x**2 for x over (-5.0, 5.0)
```

Multiple plots with single range.

```
>>> plot(x, x**2, x**3, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
```

Multiple plots with different ranges.

```
>>> plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

No adaptive sampling.

```
>>> plot(x**2, adaptive=False, nb_of_points=400)
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
```

Usage

Single Plot

`plot(expr, range, **kwargs)`

If the range is not specified, then a default range of $(-10, 10)$ is used.

Multiple plots with same range.

`plot(expr1, expr2, ..., range, **kwargs)`

If the range is not specified, then a default range of $(-10, 10)$ is used.

Multiple plots with different ranges.

`plot((expr1, range), (expr2, range), ..., **kwargs)`

Range has to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

`expr` : Expression representing the function of single variable

range: (x, 0, 5), A 3-tuple denoting the range of the free variable.

Keyword Arguments

Arguments for `plot` function:

`show`: Boolean. The default value is set to `True`. Set `show` to `False` and the function will not display the plot. The returned instance of the `Plot` class can then be used to save or display the plot by calling the `save()` and `show()` methods respectively.

Arguments for `LineOver1DRangeSeries` class:

`adaptive`: Boolean. The default value is set to `True`. Set `adaptive` to `False` and specify `nb_of_points` if uniform sampling is required.

`depth`: int Recursion depth of the adaptive algorithm. A depth of value n samples a maximum of 2^n points.

`nb_of_points`: int. Used when the `adaptive` is set to `False`. The function is uniformly sampled at `nb_of_points` number of points.

Aesthetics options:

`line_color`: float. Specifies the color for the plot. See `Plot` to see how to set color for the plots.

If there are multiple plots, then the same series series are applied to all the plots. If you want to set these options separately, you can index the `Plot` object returned and set it.

Arguments for `Plot` class:

`title`: str. Title of the plot. It is set to the latex representation of the expression, if the plot has only one expression.

`xlabel`: str. Label for the x-axis.

`ylabel`: str. Label for the y-axis.

`xscale`: {'linear', 'log'} Sets the scaling of the x-axis.

`yscale`: {'linear', 'log'} Sets the scaling if the y-axis.

`axis_center`: tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

`xlim`: tuple of two floats, denoting the x-axis limits.

`ylim`: tuple of two floats, denoting the y-axis limits.

`sympy.plotting.plot.plot_parametric(*args, **kwargs)`

Plots a 2D parametric plot.

The plotting uses an adaptive algorithm which samples recursively to accurately plot the plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

See also:

[Plot](#) (page 1059), [Parametric2DLineSeries](#) (page 1071)

Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot_parametric
>>> u = symbols('u')
```

Single Parametric plot

```
>>> plot_parametric(cos(u), sin(u), (u, -5, 5))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
```

Multiple parametric plot with single range.

```
>>> plot_parametric((cos(u), sin(u)), (u, cos(u)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-10.0, 10.0)
[1]: parametric cartesian line: (u, cos(u)) for u over (-10.0, 10.0)
```

Multiple parametric plots.

```
>>> plot_parametric((cos(u), sin(u), (u, -5, 5)),
...                 (cos(u), u, (u, -5, 5)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
[1]: parametric cartesian line: (cos(u), u) for u over (-5.0, 5.0)
```

Usage

Single plot.

`plot_parametric(expr_x, expr_y, range, **kwargs)`

If the range is not specified, then a default range of $(-10, 10)$ is used.

Multiple plots with same range.

`plot_parametric((expr1_x, expr1_y), (expr2_x, expr2_y), range, **kwargs)`

If the range is not specified, then a default range of $(-10, 10)$ is used.

Multiple plots with different ranges.

`plot_parametric((expr_x, expr_y, range), ..., **kwargs)`

Range has to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

`expr_x`: Expression representing the function along x .

`expr_y`: Expression representing the function along y .

`range`: $(u, 0, 5)$, A 3-tuple denoting the range of the parameter variable.

Keyword Arguments

Arguments for Parametric2DLineSeries class:

adaptive: Boolean. The default value is set to True. Set adaptive to False and specify **nb_of_points** if uniform sampling is required.

depth: int Recursion depth of the adaptive algorithm. A depth of value n samples a maximum of 2^n points.

nb_of_points: int. Used when the **adaptive** is set to False. The function is uniformly sampled at **nb_of_points** number of points.

Aesthetics

line_color: function which returns a float. Specifies the color for the plot. See `sympy.plotting.Plot` for more details.

If there are multiple plots, then the same Series arguments are applied to all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class:

xlabel: str. Label for the x-axis.

ylabel: str. Label for the y-axis.

xscale: {'linear', 'log'} Sets the scaling of the x-axis.

yscale: {'linear', 'log'} Sets the scaling if the y-axis.

axis_center: tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

xlim: tuple of two floats, denoting the x-axis limits.

ylim: tuple of two floats, denoting the y-axis limits.

`sympy.plotting.plot.plot3d(*args, **kwargs)`

Plots a 3D surface plot.

See also:

[Plot](#) (page 1059), [SurfaceOver2DRangeSeries](#) (page 1072)

Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot3d
>>> x, y = symbols('x y')
```

Single plot

```
>>> plot3d(x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with same range

```
>>> plot3d(x*y, -x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: -x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with different ranges.

```
>>> plot3d((x**2 + y**2, (x, -5, 5), (y, -5, 5)),
...        (x*y, (x, -3, 3), (y, -3, 3)))
Plot object containing:
[0]: cartesian surface: x**2 + y**2 for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: x*y for x over (-3.0, 3.0) and y over (-3.0, 3.0)
```

Usage

Single plot

`plot3d(expr, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of $(-10, 10)$ is used.

Multiple plot with the same range.

`plot3d(expr1, expr2, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of $(-10, 10)$ is used.

Multiple plots with different ranges.

`plot3d((expr1, range_x, range_y), (expr2, range_x, range_y), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

expr : Expression representing the function along x .

range_x: $(x, 0, 5)$, A 3-tuple denoting the range of the x variable.

range_y: $(y, 0, 5)$, A 3-tuple denoting the range of the y variable.

Keyword Arguments

Arguments for `SurfaceOver2DRangeSeries` class:

nb_of_points_x: int. The x range is sampled uniformly at `nb_of_points_x` of points.

nb_of_points_y: int. The y range is sampled uniformly at `nb_of_points_y` of points.

Aesthetics:

surface_color: Function which returns a float. Specifies the color for the surface of the plot. See `sympy.plotting.Plot` for more details.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class:

`title`: str. Title of the plot.

`sympy.plotting.plot.plot3d_parametric_line(*args, **kwargs)`
Plots a 3D parametric line plot.

See also:

[Plot](#) (page 1059), [Parametric3DLineSeries](#) (page 1072)

Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_line
>>> u = symbols('u')
```

Single plot.

```
>>> plot3d_parametric_line(cos(u), sin(u), u, (u, -5, 5))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
```

Multiple plots.

```
>>> plot3d_parametric_line((cos(u), sin(u), u, (u, -5, 5)),
... (sin(u), u**2, u, (u, -5, 5)))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
[1]: 3D parametric cartesian line: (sin(u), u**2, u) for u over (-5.0, 5.0)
```

Usage

Single plot:

`plot3d_parametric_line(expr_x, expr_y, expr_z, range, **kwargs)`

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots.

`plot3d_parametric_line((expr_x, expr_y, expr_z, range), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

`expr_x`: Expression representing the function along x.

`expr_y`: Expression representing the function along y.

`expr_z`: Expression representing the function along z.

range: (u, 0, 5), A 3-tuple denoting the range of the parameter variable.

Keyword Arguments

Arguments for Parametric3DLineSeries class.

nb_of_points: The range is uniformly sampled at nb_of_points number of points.

Aesthetics:

line_color: function which returns a float. Specifies the color for the plot. See `sympy.plotting.Plot` for more details.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned `Plot` object and set it.

Arguments for `Plot` class.

title : str. Title of the plot.

`sympy.plotting.plot.plot3d_parametric_surface(*args, **kwargs)`

Plots a 3D parametric surface plot.

See also:

[Plot](#) (page 1059), [ParametricSurfaceSeries](#) (page 1072)

Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_surface
>>> u, v = symbols('u v')
```

Single plot.

```
>>> plot3d_parametric_surface(cos(u + v), sin(u - v), u - v,
...                             (u, -5, 5), (v, -5, 5))
Plot object containing:
[0]: parametric cartesian surface: (cos(u + v), sin(u - v), u - v) for u over (-5.
→0, 5.0) and v over (-5.0, 5.0)
```

Usage

Single plot.

`plot3d_parametric_surface(expr_x, expr_y, expr_z, range_u, range_v, **kwargs)`

If the ranges is not specified, then a default range of (-10, 10) is used.

Multiple plots.

`plot3d_parametric_surface((expr_x, expr_y, expr_z, range_u, range_v), ... , **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

`expr_x`: Expression representing the function along x .

`expr_y`: Expression representing the function along y .

`expr_z`: Expression representing the function along z .

`range_u`: $(u, 0, 5)$, A 3-tuple denoting the range of the u variable.

`range_v`: $(v, 0, 5)$, A 3-tuple denoting the range of the v variable.

Keyword Arguments

Arguments for `ParametricSurfaceSeries` class:

`nb_of_points_u`: int. The u range is sampled uniformly at `nb_of_points_v` of points

`nb_of_points_y`: int. The v range is sampled uniformly at `nb_of_points_y` of points

Aesthetics:

`surface_color`: Function which returns a float. Specifies the color for the surface of the plot. See `sympy.plotting.Plot` for more details.

If there are multiple plots, then the same series arguments are applied for all the plots. If you want to set these options separately, you can index the returned `Plot` object and set it.

Arguments for `Plot` class:

`title`: str. Title of the plot.

`sympy.plotting.plot_implicit.plot_implicit(expr, x_var=None, y_var=None, **kwargs)`

A plot function to plot implicit equations / inequalities.

Examples

Plot expressions:

```
>>> from sympy import plot_implicit, cos, sin, symbols, Eq, And
>>> x, y = symbols('x y')
```

Without any ranges for the symbols in the expression

```
>>> p1 = plot_implicit(Eq(x**2 + y**2, 5))
```

With the range for the symbols

```
>>> p2 = plot_implicit(Eq(x**2 + y**2, 3),
...                    (x, -3, 3), (y, -3, 3))
```

With depth of recursion as argument.

```
>>> p3 = plot_implicit(Eq(x**2 + y**2, 5),
...                    (x, -4, 4), (y, -4, 4), depth = 2)
```

Using mesh grid and not using adaptive meshing.

```
>>> p4 = plot_implicit(Eq(x**2 + y**2, 5),  
...                   (x, -5, 5), (y, -2, 2), adaptive=False)
```

Using mesh grid with number of points as input.

```
>>> p5 = plot_implicit(Eq(x**2 + y**2, 5),  
...                   (x, -5, 5), (y, -2, 2),  
...                   adaptive=False, points=400)
```

Plotting regions.

```
>>> p6 = plot_implicit(y > x**2)
```

Plotting Using boolean conjunctions.

```
>>> p7 = plot_implicit(And(y > x, y > -x))
```

When plotting an expression with a single variable ($y - 1$, for example), specify the x or the y variable explicitly:

```
>>> p8 = plot_implicit(y - 1, y_var=y)  
>>> p9 = plot_implicit(x - 1, x_var=x)
```

Arguments

- `expr` : The equation / inequality that is to be plotted.
- `x_var` (optional) : symbol to plot on x-axis or tuple giving symbol and range as (symbol, xmin, xmax)
- `y_var` (optional) : symbol to plot on y-axis or tuple giving symbol and range as (symbol, ymin, ymax)

If neither `x_var` nor `y_var` are given then the free symbols in the expression will be assigned in the order they are sorted.

The following keyword arguments can also be used:

- **adaptive**. **Boolean. The default value is set to True. It has to be** set to False if you want to use a mesh grid.
- **depth integer. The depth of recursion for adaptive mesh grid.** Default value is 0. Takes value in the range (0, 4).
- **points integer. The number of points if adaptive mesh grid is not** used. Default value is 200.
- `title` string. The title for the plot.
- `xlabel` string. The label for the x-axis
- `ylabel` string. The label for the y-axis

Aesthetics options:

- **line_color: float or string. Specifies the color for the plot.** See Plot to see how to set color for the plots.

`plot_implicit`, by default, uses interval arithmetic to plot functions. If the expression cannot be plotted using interval arithmetic, it defaults to a generating a contour using a mesh grid of fixed number of points. By setting `adaptive` to False, you can force `plot_implicit`

to use the mesh grid. The mesh grid method can be effective when adaptive plotting using interval arithmetic, fails to plot with small line width.

5.20.4 Series Classes

class sympy.plotting.plot.BaseSeries

Base class for the data objects containing stuff to be plotted.

The backend should check if it supports the data series that it's given. (eg TextBackend supports only LineOver1DRange). It's the backend responsibility to know how to use the class of data series that it's given.

Some data series classes are grouped (using a class attribute like is_2Dline) according to the api they present (based only on convention). The backend is not obliged to use that api (eg. The LineOver1DRange belongs to the is_2Dline group and presents the get_points method, but the TextBackend does not use the get_points method).

class sympy.plotting.plot.Line2DBaseSeries

A base class for 2D lines.

- adding the label, steps and only_integers options
- making is_2Dline true
- defining get_segments and get_color_array

class sympy.plotting.plot.LineOver1DRangeSeries(expr, var_start_end, **kwargs)

Representation for a line consisting of a SymPy expression over a range.

get_segments()

Adaptively gets segments for plotting.

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

- [1] **Adaptive polygonal approximation of parametric curves**, Luiz Henrique de Figueiredo.

class sympy.plotting.plot.Parametric2DLineSeries(expr_x, expr_y, var_start_end, **kwargs)

Representation for a line consisting of two parametric sympy expressions over a range.

get_segments()

Adaptively gets segments for plotting.

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

- [1] **Adaptive polygonal approximation of parametric curves**, Luiz Henrique de Figueiredo.

class sympy.plotting.plot.Line3DBaseSeries

A base class for 3D lines.

Most of the stuff is derived from Line2DBaseSeries.

class sympy.plotting.plot.Parametric3DLineSeries(expr_x, expr_y, expr_z,
var_start_end, **kwargs)

Representation for a 3D line consisting of two parametric sympy expressions and a range.

class sympy.plotting.plot.SurfaceBaseSeries

A base class for 3D surfaces.

class sympy.plotting.plot.SurfaceOver2DRangeSeries(expr, var_start_end_x,
var_start_end_y, **kwargs)

Representation for a 3D surface consisting of a sympy expression and 2D range.

class sympy.plotting.plot.ParametricSurfaceSeries(expr_x, expr_y, expr_z,
var_start_end_u,
var_start_end_v, **kwargs)

Representation for a 3D surface consisting of three parametric sympy expressions and a range.

class sympy.plotting.plot_implicit.ImplicitSeries(expr, var_start_end_x,
var_start_end_y,
has_equality,
use_interval_math, depth,
nb_of_points, line_color)

Representation for Implicit plot

5.21 Pyglet Plotting Module

This is the documentation for the old plotting module that uses pyglet. This module has some limitations and is not actively developed anymore. For an alternative you can look at the new plotting module.

The pyglet plotting module can do nice 2D and 3D plots that can be controlled by console commands as well as keyboard and mouse, with the only dependency being pyglet.

Here is the simplest usage:

```
>>> from sympy import var, Plot
>>> var('x y z')
>>> Plot(x*y**3-y*x**3)
```

To see lots of plotting examples, see examples/pyglet_plotting.py and try running it in interactive mode (python -i plotting.py):

```
$ python -i examples/pyglet_plotting.py
```

And type for instance example(7) or example(11).

See also the [Plotting Module](#) wiki page for screenshots.

5.21.1 Plot Window Controls

Camera	Keys
Sensitivity Modifier	SHIFT
Zoom	R and F, Page Up and Down, Numpad + and -
Rotate View X,Y axis	Arrow Keys, A,S,D,W, Numpad 4,6,8,2
Rotate View Z axis	Q and E, Numpad 7 and 9
Rotate Ordinate Z axis	Z and C, Numpad 1 and 3
View XY	F1
View XZ	F2
View YZ	F3
View Perspective	F4
Reset	X, Numpad 5

Axes	Keys
Toggle Visible	F5
Toggle Colors	F6

Window	Keys
Close	ESCAPE
Screenshot	F8

The mouse can be used to rotate, zoom, and translate by dragging the left, middle, and right mouse buttons respectively.

5.21.2 Coordinate Modes

Plot supports several curvilinear coordinate modes, and they are independent for each plotted function. You can specify a coordinate mode explicitly with the 'mode' named argument, but it can be automatically determined for cartesian or parametric plots, and therefore must only be specified for polar, cylindrical, and spherical modes.

Specifically, `Plot(function arguments)` and `Plot.__setitem__(i, function arguments)` (accessed using array-index syntax on the Plot instance) will interpret your arguments as a cartesian plot if you provide one function and a parametric plot if you provide two or three functions. Similarly, the arguments will be interpreted as a curve if one variable is used, and a surface if two are used.

Supported mode names by number of variables:

- 1 (curves): parametric, cartesian, polar
- 2 (surfaces): parametric, cartesian, cylindrical, spherical

```
>>> Plot(1, 'mode=spherical; color=zfade4')
```

Note that function parameters are given as option strings of the form "key1=value1; key2 = value2" (spaces are truncated). Keyword arguments given directly to plot apply to the plot itself.

5.21.3 Specifying Intervals for Variables

The basic format for variable intervals is [var, min, max, steps]. However, the syntax is quite flexible, and arguments not specified are taken from the defaults for the current coordinate mode:

```
>>> Plot(x**2) # implies [x,-5,5,100]
>>> Plot(x**2, [], []) # [x,-1,1,40], [y,-1,1,40]
>>> Plot(x**2-y**2, [100], [100]) # [x,-1,1,100], [y,-1,1,100]
>>> Plot(x**2, [x,-13,13,100])
>>> Plot(x**2, [-13,13]) # [x,-13,13,100]
>>> Plot(x**2, [x,-13,13]) # [x,-13,13,100]
>>> Plot(1*x, [], [x], 'mode=cylindrical') # [unbound_theta,0,2*Pi,40], [x,-1,1,20]
```

5.21.4 Using the Interactive Interface

```
>>> p = Plot(visible=False)
>>> f = x**2
>>> p[1] = f
>>> p[2] = f.diff(x)
>>> p[3] = f.diff(x).diff(x)
>>> p
[1]: x**2, 'mode=cartesian'
[2]: 2*x, 'mode=cartesian'
[3]: 2, 'mode=cartesian'
>>> p.show()
>>> p.clear()
>>> p
<blank plot>
>>> p[1] = x**2+y**2
>>> p[1].style = 'solid'
>>> p[2] = -x**2-y**2
>>> p[2].style = 'wireframe'
>>> p[1].color = z, (0.4,0.4,0.9), (0.9,0.4,0.4)
>>> p[1].style = 'both'
>>> p[2].style = 'both'
>>> p.close()
```

5.21.5 Using Custom Color Functions

The following code plots a saddle and color it by the magnitude of its gradient:

```
>>> fz = x**2-y**2
>>> Fx, Fy, Fz = fz.diff(x), fz.diff(y), 0
>>> p[1] = fz, 'style=solid'
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5)
```

The coloring algorithm works like this:

1. Evaluate the color function(s) across the curve or surface.
2. Find the minimum and maximum value of each component.
3. Scale each component to the color gradient.

When not specified explicitly, the default color gradient is $f(0.0)=(0.4,0.4,0.4) \rightarrow f(1.0)=(0.9,0.9,0.9)$. In our case, everything is gray-scale because we have applied the default color gradient uniformly for each color component. When defining a color scheme in this way, you might want to supply a color gradient as well:

```
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5), (0.1,0.1,0.9), (0.9,0.1,0.1)
```

Here's a color gradient with four steps:

```
>>> gradient = [ 0.0, (0.1,0.1,0.9), 0.3, (0.1,0.9,0.1),
...              0.7, (0.9,0.9,0.1), 1.0, (1.0,0.0,0.0) ]
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5), gradient
```

The other way to specify a color scheme is to give a separate function for each component r, g, b. With this syntax, the default color scheme is defined:

```
>>> p[1].color = z,y,x, (0.4,0.4,0.4), (0.9,0.9,0.9)
```

This maps $z \rightarrow \text{red}$, $y \rightarrow \text{green}$, and $x \rightarrow \text{blue}$. In some cases, you might prefer to use the following alternative syntax:

```
>>> p[1].color = z,(0.4,0.9), y,(0.4,0.9), x,(0.4,0.9)
```

You can still use multi-step gradients with three-function color schemes.

5.21.6 Plotting Geometric Entities

The plotting module is capable of plotting some 2D geometric entities like line, circle and ellipse. The following example plots a circle and a tangent line at a random point on the ellipse.

```
In [1]: p = Plot(axes='label_axes=True')
In [2]: c = Circle(Point(0,0), 1)
In [3]: t = c.tangent_line(c.random_point())
In [4]: p[0] = c
In [5]: p[1] = t
```

Plotting polygons (Polygon, RegularPolygon, Triangle) are not supported directly. However a polygon can be plotted through a loop as follows.

```
In [6]: p = Plot(axes='label_axes=True')
In [7]: t = RegularPolygon(Point(0,0), 1, 5)
In [8]: for i in range(len(t.sides)):
...:     p[i] = t.sides[i]
```

5.22 Assumptions module

5.22.1 Contents

Ask

Module for querying SymPy objects about assumptions.

class sympy.assumptions.ask.AssumptionKeys

This class contains all the supported keys by ask.

algebraic

Algebraic number predicate.

`Q.algebraic(x)` is true iff x belongs to the set of algebraic numbers. x is algebraic if there is some polynomial in $p(x) \in \mathbb{Q}[x]$ such that $p(x) = 0$.

References

[R1] (page 1867)

Examples

```
>>> from sympy import ask, Q, sqrt, I, pi
>>> ask(Q.algebraic(sqrt(2)))
True
>>> ask(Q.algebraic(I))
True
>>> ask(Q.algebraic(pi))
False
```

antihermitian

Antihermitian predicate.

`Q.antihermitian(x)` is true iff x belongs to the field of antihermitian operators, i.e., operators in the form $x*I$, where x is Hermitian.

References

[R2] (page 1867)

bounded

See documentation of `Q.finite`.

commutative

Commutative predicate.

`ask(Q.commutative(x))` is true iff x commutes with any other object with respect to multiplication operation.

complex

Complex number predicate.

`Q.complex(x)` is true iff x belongs to the set of complex numbers. Note that every complex number is finite.

References

[R3] (page 1868)

Examples

```
>>> from sympy import Q, Symbol, ask, I, oo
>>> x = Symbol('x')
>>> ask(Q.complex(0))
True
>>> ask(Q.complex(2 + 3*I))
True
>>> ask(Q.complex(oo))
False
```

complex_elements

Complex elements matrix predicate.

`Q.complex_elements(x)` is true iff all the elements of `x` are complex numbers.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.complex(X[1, 2]), Q.complex_elements(X))
True
>>> ask(Q.complex_elements(X), Q.integer_elements(X))
True
```

composite

Composite number predicate.

`ask(Q.composite(x))` is true iff `x` is a positive integer and has at least one positive divisor other than 1 and the number itself.

Examples

```
>>> from sympy import Q, ask
>>> ask(Q.composite(0))
False
>>> ask(Q.composite(1))
False
>>> ask(Q.composite(2))
False
>>> ask(Q.composite(20))
True
```

diagonal

Diagonal matrix predicate.

`Q.diagonal(x)` is true iff `x` is a diagonal matrix. A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero.

References

[R4] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.diagonal(ZeroMatrix(3, 3)))
True
>>> ask(Q.diagonal(X), Q.lower_triangular(X) &
...     Q.upper_triangular(X))
True
```

even

Even number predicate.

`ask(Q.even(x))` is true iff x belongs to the set of even integers.

Examples

```
>>> from sympy import Q, ask, pi
>>> ask(Q.even(0))
True
>>> ask(Q.even(2))
True
>>> ask(Q.even(3))
False
>>> ask(Q.even(pi))
False
```

extended_real

Extended real predicate.

`Q.extended_real(x)` is true iff x is a real number or $\{-\infty, \infty\}$.

See documentation of `Q.real` for more information about related facts.

Examples

```
>>> from sympy import ask, Q, oo, I
>>> ask(Q.extended_real(1))
True
>>> ask(Q.extended_real(I))
False
>>> ask(Q.extended_real(oo))
True
```

finite

Finite predicate.

`Q.finite(x)` is true if x is neither an infinity nor a NaN. In other words, `ask(Q.finite(x))` is true for all x having a bounded absolute value.

References

[R5] (page 1868)

Examples

```
>>> from sympy import Q, ask, Symbol, S, oo, I
>>> x = Symbol('x')
>>> ask(Q.finite(S.NaN))
False
>>> ask(Q.finite(oo))
False
>>> ask(Q.finite(1))
True
>>> ask(Q.finite(2 + 3*I))
True
```

fullrank

Fullrank matrix predicate.

`Q.fullrank(x)` is true iff `x` is a full rank matrix. A matrix is full rank if all rows and columns of the matrix are linearly independent. A square matrix is full rank iff its determinant is nonzero.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.fullrank(X.T), Q.fullrank(X))
True
>>> ask(Q.fullrank(ZeroMatrix(3, 3)))
False
>>> ask(Q.fullrank(Identity(3)))
True
```

hermitian

Hermitian predicate.

`ask(Q.hermitian(x))` is true iff `x` belongs to the set of Hermitian operators.

References

[R6] (page 1868)

imaginary

Imaginary number predicate.

`Q.imaginary(x)` is true iff `x` can be written as a real number multiplied by the imaginary unit `I`. Please note that `0` is not considered to be an imaginary number.

References

[R7] (page 1868)

Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.imaginary(3*I))
True
>>> ask(Q.imaginary(2 + 3*I))
False
>>> ask(Q.imaginary(0))
False
```

infinite

Infinite number predicate.

`Q.infinite(x)` is true iff the absolute value of `x` is infinity.

infinitesimal

See documentation of `Q.zero`.

infinity

See documentation of `Q.infinite`.

integer

Integer predicate.

`Q.integer(x)` is true iff `x` belongs to the set of integer numbers.

References

[R8] (page 1868)

Examples

```
>>> from sympy import Q, ask, S
>>> ask(Q.integer(5))
True
>>> ask(Q.integer(S(1)/2))
False
```

integer_elements

Integer elements matrix predicate.

`Q.integer_elements(x)` is true iff all the elements of `x` are integers.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.integer(X[1, 2]), Q.integer_elements(X))
True
```

invertible

Invertible matrix predicate.

`Q.invertible(x)` is true iff `x` is an invertible matrix. A square matrix is called invertible only if its determinant is not 0.

References

[R9] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.invertible(X*Y), Q.invertible(X))
False
>>> ask(Q.invertible(X*Z), Q.invertible(X) & Q.invertible(Z))
True
>>> ask(Q.invertible(X), Q.fullrank(X) & Q.square(X))
True
```

irrational

Irrational number predicate.

`Q.irrational(x)` is true iff x is any real number that cannot be expressed as a ratio of integers.

References

[R10] (page 1868)

Examples

```
>>> from sympy import ask, Q, pi, S, I
>>> ask(Q.irrational(0))
False
>>> ask(Q.irrational(S(1)/2))
False
>>> ask(Q.irrational(pi))
True
>>> ask(Q.irrational(I))
False
```

is_true

Generic predicate.

`ask(Q.is_true(x))` is true iff x is true. This only makes sense if x is a predicate.

Examples

```
>>> from sympy import ask, Q, symbols
>>> x = symbols('x')
>>> ask(Q.is_true(True))
True
```

lower_triangular

Lower triangular matrix predicate.

A matrix M is called lower triangular matrix if $a_{ij} = 0$ for $i > j$.

References

[R11] (page 1868)

Examples

```
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.lower_triangular(Identity(3)))
True
>>> ask(Q.lower_triangular(ZeroMatrix(3, 3)))
True
```

negative

Negative number predicate.

$Q.negative(x)$ is true iff x is a real number and $x < 0$, that is, it is in the interval $(-\infty, 0)$. Note in particular that negative infinity is not negative.

A few important facts about negative numbers:

- Note that $Q.nonnegative$ and $\sim Q.negative$ are not the same thing. $\sim Q.negative(x)$ simply means that x is not negative, whereas $Q.nonnegative(x)$ means that x is real and not negative, i.e., $Q.nonnegative(x)$ is logically equivalent to $Q.zero(x) \mid Q.positive(x)$. So for example, $\sim Q.negative(I)$ is true, whereas $Q.nonnegative(I)$ is false.
- See the documentation of $Q.real$ for more information about related facts.

Examples

```
>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.negative(x), Q.real(x) & ~Q.positive(x) & ~Q.zero(x))
True
>>> ask(Q.negative(-1))
True
>>> ask(Q.nonnegative(I))
False
>>> ask(~Q.negative(I))
True
```

nonnegative

Nonnegative real number predicate.

$ask(Q.nonnegative(x))$ is true iff x belongs to the set of positive numbers including zero.

- Note that $Q.nonnegative$ and $\sim Q.negative$ are not the same thing. $\sim Q.negative(x)$ simply means that x is not negative, whereas $Q.nonnegative(x)$

means that x is real and not negative, i.e., $Q.\text{nonnegative}(x)$ is logically equivalent to $Q.\text{zero}(x) \mid Q.\text{positive}(x)$. So for example, $\sim Q.\text{negative}(I)$ is true, whereas $Q.\text{nonnegative}(I)$ is false.

Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.nonnegative(1))
True
>>> ask(Q.nonnegative(0))
True
>>> ask(Q.nonnegative(-1))
False
>>> ask(Q.nonnegative(I))
False
>>> ask(Q.nonnegative(-I))
False
```

nonpositive

Nonpositive real number predicate.

$\text{ask}(Q.\text{nonpositive}(x))$ is true iff x belongs to the set of negative numbers including zero.

- Note that $Q.\text{nonpositive}$ and $\sim Q.\text{positive}$ are not the same thing. $\sim Q.\text{positive}(x)$ simply means that x is not positive, whereas $Q.\text{nonpositive}(x)$ means that x is real and not positive, i.e., $Q.\text{nonpositive}(x)$ is logically equivalent to $Q.\text{negative}(x) \mid Q.\text{zero}(x)$. So for example, $\sim Q.\text{positive}(I)$ is true, whereas $Q.\text{nonpositive}(I)$ is false.

Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.nonpositive(-1))
True
>>> ask(Q.nonpositive(0))
True
>>> ask(Q.nonpositive(1))
False
>>> ask(Q.nonpositive(I))
False
>>> ask(Q.nonpositive(-I))
False
```

nonzero

Nonzero real number predicate.

$\text{ask}(Q.\text{nonzero}(x))$ is true iff x is real and x is not zero. Note in particular that $Q.\text{nonzero}(x)$ is false if x is not real. Use $\sim Q.\text{zero}(x)$ if you want the negation of being zero without any real assumptions.

A few important facts about nonzero numbers:

- $Q.\text{nonzero}$ is logically equivalent to $Q.\text{positive} \mid Q.\text{negative}$.
- See the documentation of $Q.\text{real}$ for more information about related facts.

Examples

```
>>> from sympy import Q, ask, symbols, I, oo
>>> x = symbols('x')
>>> print(ask(Q.nonzero(x), ~Q.zero(x)))
None
>>> ask(Q.nonzero(x), Q.positive(x))
True
>>> ask(Q.nonzero(x), Q.zero(x))
False
>>> ask(Q.nonzero(0))
False
>>> ask(Q.nonzero(I))
False
>>> ask(~Q.zero(I))
True
>>> ask(Q.nonzero(oo))
False
```

normal

Normal matrix predicate.

A matrix is normal if it commutes with its conjugate transpose.

References

[R12] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.normal(X), Q.unitary(X))
True
```

odd

Odd number predicate.

`ask(Q.odd(x))` is true iff `x` belongs to the set of odd numbers.

Examples

```
>>> from sympy import Q, ask, pi
>>> ask(Q.odd(0))
False
>>> ask(Q.odd(2))
False
>>> ask(Q.odd(3))
True
>>> ask(Q.odd(pi))
False
```

orthogonal

Orthogonal matrix predicate.

`Q.orthogonal(x)` is true iff x is an orthogonal matrix. A square matrix M is an orthogonal matrix if it satisfies $M^T M = M M^T = I$ where M^T is the transpose matrix of M and I is an identity matrix. Note that an orthogonal matrix is necessarily invertible.

References

[R13] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.orthogonal(Y))
False
>>> ask(Q.orthogonal(X*Z*X), Q.orthogonal(X) & Q.orthogonal(Z))
True
>>> ask(Q.orthogonal(Identity(3)))
True
>>> ask(Q.invertible(X), Q.orthogonal(X))
True
```

positive

Positive real number predicate.

`Q.positive(x)` is true iff x is real and $x > 0$, that is if x is in the interval $(0, \infty)$. In particular, infinity is not positive.

A few important facts about positive numbers:

- Note that `Q.nonpositive` and `~Q.positive` are not the same thing. `~Q.positive(x)` simply means that x is not positive, whereas `Q.nonpositive(x)` means that x is real and not positive, i.e., `Q.nonpositive(x)` is logically equivalent to `Q.negative(x)|Q.zero(x)`. So for example, `~Q.positive(I)` is true, whereas `Q.nonpositive(I)` is false.
- See the documentation of `Q.real` for more information about related facts.

Examples

```
>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.positive(x), Q.real(x) & ~Q.negative(x) & ~Q.zero(x))
True
>>> ask(Q.positive(1))
True
>>> ask(Q.nonpositive(I))
False
>>> ask(~Q.positive(I))
True
```

positive_definite

Positive definite matrix predicate.

If M is a $n \times n$ symmetric real matrix, it is said to be positive definite if $Z^T M Z$ is positive for every non-zero column vector Z of n real numbers.

References

[R14] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.positive_definite(Y))
False
>>> ask(Q.positive_definite(Identity(3)))
True
>>> ask(Q.positive_definite(X + Z), Q.positive_definite(X) &
...     Q.positive_definite(Z))
True
```

prime

Prime number predicate.

`ask(Q.prime(x))` is true iff x is a natural number greater than 1 that has no positive divisors other than 1 and the number itself.

Examples

```
>>> from sympy import Q, ask
>>> ask(Q.prime(0))
False
>>> ask(Q.prime(1))
False
>>> ask(Q.prime(2))
True
>>> ask(Q.prime(20))
False
>>> ask(Q.prime(-3))
False
```

rational

Rational number predicate.

`Q.rational(x)` is true iff x belongs to the set of rational numbers.

References

https://en.wikipedia.org/wiki/Rational_number

Examples

```
>>> from sympy import ask, Q, pi, S
>>> ask(Q.rational(0))
True
>>> ask(Q.rational(S(1)/2))
True
>>> ask(Q.rational(pi))
False
```

real

Real number predicate.

`Q.real(x)` is true iff x is a real number, i.e., it is in the interval $(-\infty, \infty)$. Note that, in particular the infinities are not real. Use `Q.extended_real` if you want to consider those as well.

A few important facts about reals:

- Every real number is positive, negative, or zero. Furthermore, because these sets are pairwise disjoint, each real number is exactly one of those three.
- Every real number is also complex.
- Every real number is finite.
- Every real number is either rational or irrational.
- Every real number is either algebraic or transcendental.
- The facts `Q.negative`, `Q.zero`, `Q.positive`, `Q.nonnegative`, `Q.nonpositive`, `Q.nonzero`, `Q.integer`, `Q.rational`, and `Q.irrational` all imply `Q.real`, as do all facts that imply those facts.
- The facts `Q.algebraic`, and `Q.transcendental` do not imply `Q.real`; they imply `Q.complex`. An algebraic or transcendental number may or may not be real.
- The “non” facts (i.e., `Q.nonnegative`, `Q.nonzero`, `Q.nonpositive` and `Q.noninteger`) are not equivalent to not the fact, but rather, not the fact and `Q.real`. For example, `Q.nonnegative` means $\sim Q.negative \ \& \ Q.real$. So for example, i is not nonnegative, nonzero, or nonpositive.

References

[R15] (page 1868)

Examples

```
>>> from sympy import Q, ask, symbols
>>> x = symbols('x')
>>> ask(Q.real(x), Q.positive(x))
True
>>> ask(Q.real(0))
True
```

real_elements

Real elements matrix predicate.

`Q.real_elements(x)` is true iff all the elements of x are real numbers.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.real(X[1, 2]), Q.real_elements(X))
True
```

singular

Singular matrix predicate.

A matrix is singular iff the value of its determinant is 0.

References

[R16] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.singular(X), Q.invertible(X))
False
>>> ask(Q.singular(X), ~Q.invertible(X))
True
```

square

Square matrix predicate.

`Q.square(x)` is true iff `x` is a square matrix. A square matrix is a matrix with the same number of rows and columns.

References

[R17] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('X', 2, 3)
>>> ask(Q.square(X))
True
>>> ask(Q.square(Y))
False
>>> ask(Q.square(ZeroMatrix(3, 3)))
True
>>> ask(Q.square(Identity(3)))
True
```

symmetric

Symmetric matrix predicate.

`Q.symmetric(x)` is true iff `x` is a square matrix and is equal to its transpose. Every square diagonal matrix is a symmetric matrix.

References

[R18] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.symmetric(X*Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(X + Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(Y))
False
```

transcendental

Transcendental number predicate.

`Q.transcendental(x)` is true iff `x` belongs to the set of transcendental numbers. A transcendental number is a real or complex number that is not algebraic.

triangular

Triangular matrix predicate.

`Q.triangular(X)` is true if `X` is one that is either lower triangular or upper triangular.

References

[R19] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.upper_triangular(X))
True
>>> ask(Q.triangular(X), Q.lower_triangular(X))
True
```

unit_triangular

Unit triangular matrix predicate.

A unit triangular matrix is a triangular matrix with 1s on the diagonal.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.unit_triangular(X))
True
```

unitary

Unitary matrix predicate.

`Q.unitary(x)` is true iff `x` is a unitary matrix. Unitary matrix is an analogue to orthogonal matrix. A square matrix `M` with complex elements is unitary if $M^T M = I$ where M^T is the conjugate transpose matrix of `M`.

References

[R20] (page 1868)

Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.unitary(Y))
False
>>> ask(Q.unitary(X*Z*X), Q.unitary(X) & Q.unitary(Z))
True
>>> ask(Q.unitary(Identity(3)))
True
```

upper_triangular

Upper triangular matrix predicate.

A matrix `M` is called upper triangular matrix if $M_{ij} = 0$ for $i < j$.

References

[R21] (page 1868)

Examples

```
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.upper_triangular(Identity(3)))
True
>>> ask(Q.upper_triangular(ZeroMatrix(3, 3)))
True
```

zero

Zero number predicate.

`ask(Q.zero(x))` is true iff the value of `x` is zero.

Examples

```
>>> from sympy import ask, Q, oo, symbols
>>> x, y = symbols('x, y')
>>> ask(Q.zero(0))
True
>>> ask(Q.zero(1/oo))
True
>>> ask(Q.zero(0*oo))
False
>>> ask(Q.zero(1))
False
>>> ask(Q.zero(x*y), Q.zero(x) | Q.zero(y))
True
```

`sympy.assumptions.ask.ask`(proposition, assumptions=True, con-
text=AssumptionsContext())
Method for inferring properties about objects.

Syntax

- `ask(proposition)`
- `ask(proposition, assumptions)`
where proposition is any boolean expression

Examples

```
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(4*x), Q.integer(x))
False
```

Remarks Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), Q.is_true(x > 0))
```

It is however a work in progress.

`sympy.assumptions.ask.ask_full_inference`(proposition, assumptions,
known_facts_cnf)
Method for inferring properties about objects.

`sympy.assumptions.ask.compute_known_facts`(known_facts, known_facts_keys)
Compute the various forms of knowledge compilation used by the assumptions system.
This function is typically applied to the results of the `get_known_facts` and `get_known_facts_keys` functions defined at the bottom of this file.

`sympy.assumptions.ask.register_handler`(key, handler)
Register a handler in the ask system. key must be a string and handler a class inheriting from `AskHandler`:

```
>>> from sympy.assumptions import register_handler, ask, Q
>>> from sympy.assumptions.handlers import AskHandler
>>> class MersenneHandler(AskHandler):
...     # Mersenne numbers are in the form 2**n - 1, n integer
...     @staticmethod
...     def Integer(expr, assumptions):
...         from sympy import log
...         return ask(Q.integer(log(expr + 1, 2)))
>>> register_handler('mersenne', MersenneHandler)
>>> ask(Q.mersenne(7))
True
```

`sympy.assumptions.ask.remove_handler(key, handler)`
Removes a handler from the ask system. Same syntax as `register_handler`

Assume

class `sympy.assumptions.assume.AppliedPredicate`
The class of expressions resulting from applying a Predicate.

Examples

```
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> Q.integer(x)
Q.integer(x)
>>> type(Q.integer(x))
<class 'sympy.assumptions.assume.AppliedPredicate'>
```

arg
Return the expression used by this assumption.

Examples

```
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> a = Q.integer(x + 1)
>>> a.arg
x + 1
```

class `sympy.assumptions.assume.AssumptionsContext`
Set representing assumptions.

This is used to represent global assumptions, but you can also use this class to create your own local assumptions contexts. It is basically a thin wrapper to Python's `set`, so see its documentation for advanced usage.

Examples

```

>>> from sympy import AppliedPredicate, Q
>>> from sympy.assumptions.assume import global_assumptions
>>> global_assumptions
AssumptionsContext()
>>> from sympy.abc import x
>>> global_assumptions.add(Q.real(x))
>>> global_assumptions
AssumptionsContext({Q.real(x)})
>>> global_assumptions.remove(Q.real(x))
>>> global_assumptions
AssumptionsContext()
>>> global_assumptions.clear()

```

add(*assumptions)
Add an assumption.

class sympy.assumptions.assume.**Predicate**

A predicate is a function that returns a boolean value.

Predicates merely wrap their argument and remain unevaluated:

```

>>> from sympy import Q, ask
>>> type(Q.prime)
<class 'sympy.assumptions.assume.Predicate'>
>>> Q.prime.name
'prime'
>>> Q.prime(7)
Q.prime(7)
>>> _.func.name
'prime'

```

To obtain the truth value of an expression containing predicates, use the function ask:

```

>>> ask(Q.prime(7))
True

```

The tautological predicate `Q.is_true` can be used to wrap other objects:

```

>>> from sympy.abc import x
>>> Q.is_true(x > 1)
Q.is_true(x > 1)

```

eval(expr, assumptions=True)
Evaluate self(expr) under the given assumptions.

This uses only direct resolution methods, not logical inference.

sympy.assumptions.assume.assuming(*assumptions)
Context manager for assumptions

Examples

```

>>> from sympy.assumptions import assuming, Q, ask
>>> from sympy.abc import x, y

```

```

>>> print(ask(Q.integer(x + y)))
None

```

```
>>> with assuming(Q.integer(x), Q.integer(y)):
...     print(ask(Q.integer(x + y)))
True
```

Refine

`sympy.assumptions.refine.refine(expr, assumptions=True)`

Simplify an expression using assumptions.

Gives the form of `expr` that would be obtained if symbols in it were replaced by explicit numerical expressions satisfying the assumptions.

Examples

```
>>> from sympy import refine, sqrt, Q
>>> from sympy.abc import x
>>> refine(sqrt(x**2), Q.real(x))
Abs(x)
>>> refine(sqrt(x**2), Q.positive(x))
x
```

`sympy.assumptions.refine.refine_Pow(expr, assumptions)`

Handler for instances of `Pow`.

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.refine import refine_Pow
>>> from sympy.abc import x, y, z
>>> refine_Pow((-1)**x, Q.real(x))
>>> refine_Pow((-1)**x, Q.even(x))
1
>>> refine_Pow((-1)**x, Q.odd(x))
-1
```

For powers of -1, even parts of the exponent can be simplified:

```
>>> refine_Pow((-1)**(x+y), Q.even(x))
(-1)**y
>>> refine_Pow((-1)**(x+y+z), Q.odd(x) & Q.odd(z))
(-1)**y
>>> refine_Pow((-1)**(x+y+2), Q.odd(x))
(-1)**(y + 1)
>>> refine_Pow((-1)**(x+3), True)
(-1)**(x + 1)
```

`sympy.assumptions.refine.refine_Relational(expr, assumptions)`

Handler for `Relational`

```
>>> from sympy.assumptions.refine import refine_Relational
>>> from sympy.assumptions.ask import Q
>>> from sympy.abc import x
>>> refine_Relational(x<0, ~Q.is_true(x<0))
False
```

`sympy.assumptions.refine.refine_abs(expr, assumptions)`

Handler for the absolute value.

Examples

```
>>> from sympy import Symbol, Q, refine, Abs
>>> from sympy.assumptions.refine import refine_abs
>>> from sympy.abc import x
>>> refine_abs(Abs(x), Q.real(x))
>>> refine_abs(Abs(x), Q.positive(x))
x
>>> refine_abs(Abs(x), Q.negative(x))
-x
```

`sympy.assumptions.refine.refine_atan2(expr, assumptions)`
 Handler for the `atan2` function

Examples

```
>>> from sympy import Symbol, Q, refine, atan2
>>> from sympy.assumptions.refine import refine_atan2
>>> from sympy.abc import x, y
>>> refine_atan2(atan2(y,x), Q.real(y) & Q.positive(x))
atan(y/x)
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.negative(x))
atan(y/x) - pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.negative(x))
atan(y/x) + pi
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.negative(x))
pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.zero(x))
pi/2
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.zero(x))
-pi/2
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.zero(x))
nan
```

Handlers

Contents

Calculus

This module contains query handlers responsible for calculus queries: `infinitesimal`, `finite`, etc.

class `sympy.assumptions.handlers.calculus.AskFiniteHandler`
 Handler for key 'finite'.

Test that an expression is bounded respect to all its variables.

Examples of usage:

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.handlers.calculus import AskFiniteHandler
>>> from sympy.abc import x
>>> a = AskFiniteHandler()
```

(continues on next page)

(continued from previous page)

```
>>> a.Symbol(x, Q.positive(x)) == None
True
>>> a.Symbol(x, Q.finite(x))
True
```

static Add(expr, assumptions)

Return True if expr is bounded, False if not and None if unknown.

Truth Table:

		B	U			?		
			'+'	'-'	'x'	'+'	'-'	'x'
B		B	U			?		
U	'+'		U	?	?	U	?	?
	'-'		?	U	?	?	U	?
	'x'		?			?		
?						?		

- 'B' = Bounded
- 'U' = Unbounded
- '?' = unknown boundedness
- '+' = positive sign
- '-' = negative sign
- 'x' = sign unknown

- All Bounded -> True
- 1 Unbounded and the rest Bounded -> False
- >1 Unbounded, all with same known sign -> False
- Any Unknown and unknown sign -> None
- Else -> None

When the signs are not the same you can have an undefined result as in oo - oo, hence 'bounded' is also undefined.

static Mul(expr, assumptions)

Return True if expr is bounded, False if not and None if unknown.

Truth Table:

	B	U	?	
			s	/s
B	B	U	?	
U		U	U	?
?			?	

- B = Bounded

- U = Unbounded
- ? = unknown boundedness
- s = signed (hence nonzero)
- /s = not signed

static Pow(expr, assumptions)
 Unbounded ** NonZero -> Unbounded Bounded ** Bounded -> Bounded Abs() $\leq$ 1
 ** Positive -> Bounded Abs() $\geq$ 1 ** Negative -> Bounded Otherwise unknown

static Symbol(expr, assumptions)
 Handles Symbol.

Examples

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.handlers.calculus import AskFiniteHandler
>>> from sympy.abc import x
>>> a = AskFiniteHandler()
>>> a.Symbol(x, Q.positive(x)) == None
True
>>> a.Symbol(x, Q.finite(x))
True
```

nTheory

Handlers for keys related to number theory: prime, even, odd, etc.

class sympy.assumptions.handlers.ntheory.**AskOddHandler**
 Handler for key 'odd' Test that an expression represents an odd number

class sympy.assumptions.handlers.ntheory.**AskPrimeHandler**
 Handler for key 'prime' Test that an expression represents a prime number. When the expression is a number the result, when True, is subject to the limitations of isprime() which is used to return the result.

static Pow(expr, assumptions)
 Integer**Integer -> !Prime

Order

AskHandlers related to order relations: positive, negative, etc.

class sympy.assumptions.handlers.order.**AskNegativeHandler**
 This is called by ask() when key='negative'
 Test that an expression is less (strict) than zero.

Examples

```
>>> from sympy import ask, Q, pi
>>> ask(Q.negative(pi+1)) # this calls AskNegativeHandler.Add
False
>>> ask(Q.negative(pi**2)) # this calls AskNegativeHandler.Pow
False
```

static Add(expr, assumptions)

Positive + Positive -> Positive, Negative + Negative -> Negative

static Pow(expr, assumptions)

Real ** Even -> NonNegative Real ** Odd -> same_as_base NonNegative ** Positive
-> NonNegative

class sympy.assumptions.handlers.order.**AskNonZeroHandler**

Handler for key 'zero' Test that an expression is not identically zero

class sympy.assumptions.handlers.order.**AskPositiveHandler**

Handler for key 'positive' Test that an expression is greater (strict) than zero

Sets

Handlers for predicates related to set membership: integer, rational, etc.

class sympy.assumptions.handlers.sets.**AskAlgebraicHandler**

Handler for Q.algebraic key.

class sympy.assumptions.handlers.sets.**AskAntiHermitianHandler**

Handler for Q.antihermitian Test that an expression belongs to the field of anti-Hermitian operators, that is, operators in the form $x*I$, where x is Hermitian

static Add(expr, assumptions)

Antihermitian + Antihermitian -> Antihermitian Antihermitian + !Antihermitian -> !Antihermitian

static Mul(expr, assumptions)

As long as there is at most only one noncommutative term: Hermitian*Hermitian
-> !Antihermitian Hermitian*Antihermitian -> Antihermitian Antihermitian*Antihermitian -> !Antihermitian

static Pow(expr, assumptions)

Hermitian**Integer -> !Antihermitian Antihermitian**Even -> !Antihermitian Antihermitian**Odd -> Antihermitian

class sympy.assumptions.handlers.sets.**AskComplexHandler**

Handler for Q.complex Test that an expression belongs to the field of complex numbers

class sympy.assumptions.handlers.sets.**AskExtendedRealHandler**

Handler for Q.extended_real Test that an expression belongs to the field of extended real numbers, that is real numbers union {Infinity, -Infinity}

class sympy.assumptions.handlers.sets.**AskHermitianHandler**

Handler for Q.hermitian Test that an expression belongs to the field of Hermitian operators

static Add(expr, assumptions)

Hermitian + Hermitian -> Hermitian Hermitian + !Hermitian -> !Hermitian

static Mul(expr, assumptions)

As long as there is at most only one noncommutative term: Hermitian*Hermitian ->


```

    Hermitian Hermitian*Antihhermitian -> !Hermitian Antihhermitian*Antihhermitian ->
    Hermitian
static Pow(expr, assumptions)
    Hermitian**Integer -> Hermitian
class sympy.assumptions.handlers.sets.AskImaginaryHandler
    Handler for Q.imaginary Test that an expression belongs to the field of imaginary num-
    bers, that is, numbers in the form  $x*I$ , where  $x$  is real
static Add(expr, assumptions)
    Imaginary + Imaginary -> Imaginary Imaginary + Complex -> ? Imaginary + Real
    -> !Imaginary
static Mul(expr, assumptions)
    Real*Imaginary -> Imaginary Imaginary*Imaginary -> Real
static Pow(expr, assumptions)
    Imaginary**Odd -> Imaginary Imaginary**Even -> Real  $b^{**Imaginary}$  -> !Imagi-
    nary if exponent is an integer multiple of  $I*pi/log(b)$  Imaginary**Real -> ? Posi-
    tive**Real -> Real Negative**Integer -> Real Negative**(Integer/2) -> Imaginary
    Negative**Real -> not Imaginary if exponent is not Rational
class sympy.assumptions.handlers.sets.AskIntegerHandler
    Handler for Q.integer Test that an expression belongs to the field of integer numbers
static Add(expr, assumptions)
    Integer + Integer -> Integer Integer + !Integer -> !Integer !Integer + !Integer -> ?
static Mul(expr, assumptions)
    Integer*Integer -> Integer Integer*Irrational -> !Integer Odd/Even -> !Integer In-
    teger*Rational -> ?
static Pow(expr, assumptions)
    Integer + Integer -> Integer Integer + !Integer -> !Integer !Integer + !Integer -> ?
class sympy.assumptions.handlers.sets.AskRationalHandler
    Handler for Q.rational Test that an expression belongs to the field of rational numbers
static Add(expr, assumptions)
    Rational + Rational -> Rational Rational + !Rational -> !Rational !Rational + !Ra-
    tional -> ?
static Mul(expr, assumptions)
    Rational + Rational -> Rational Rational + !Rational -> !Rational !Rational + !Ra-
    tional -> ?
static Pow(expr, assumptions)
    Rational ** Integer -> Rational Irrational ** Rational -> Irrational Rational ** Irra-
    tional -> ?
class sympy.assumptions.handlers.sets.AskRealHandler
    Handler for Q.real Test that an expression belongs to the field of real numbers
static Add(expr, assumptions)
    Real + Real -> Real Real + (Complex & !Real) -> !Real
static Mul(expr, assumptions)
    Real*Real -> Real Real*Imaginary -> !Real Imaginary*Imaginary -> Real
static Pow(expr, assumptions)
    Real**Integer -> Real Positive**Real -> Real Real**(Integer/Even) -> Real if base is
    nonnegative Real**(Integer/Odd) -> Real Imaginary**(Integer/Even) -> Real Imagi-
    nary**(Integer/Odd) -> not Real Imaginary**Real -> ? since Real could be 0 (giving

```

real) or 1 (giving imaginary) $b^{**}\text{Imaginary} \rightarrow \text{Real}$ if $\log(b)$ is imaginary and $b \neq 0$ and exponent $\neq$ integer multiple of $I*\pi/\log(b)$ $\text{Real}^{**}\text{Real} \rightarrow ?$ e.g. $\text{sqrt}(-1)$ is imaginary and $\text{sqrt}(2)$ is not

Queries are used to ask information about expressions. Main method for this is `ask()`:

```
sympy.assumptions.ask.ask(proposition, assumptions=True, con-
                           text=AssumptionsContext())
```

Method for inferring properties about objects.

Syntax

- `ask(proposition)`
- `ask(proposition, assumptions)`
where `proposition` is any boolean expression

Examples

```
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(4*x), Q.integer(x))
False
```

Remarks Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), Q.is_true(x > 0))
```

It is however a work in progress.

5.22.2 Querying

`ask`'s optional second argument should be a boolean expression involving assumptions about objects in `expr`. Valid values include:

- `Q.integer(x)`
- `Q.positive(x)`
- `Q.integer(x) & Q.positive(x)`
- etc.

`Q` is an object holding known predicates.

See documentation for the logic module for a complete list of valid boolean expressions.

You can also define a context so you don't have to pass that argument each time to function `ask()`. This is done by using the assuming context manager from module `sympy.assumptions`.

```
>>> from sympy import *
>>> x = Symbol('x')
>>> y = Symbol('y')
```

(continues on next page)

(continued from previous page)

```
>>> facts = Q.positive(x), Q.positive(y)
>>> with assuming(*facts):
...     print(ask(Q.positive(2*x + y)))
True
```

5.22.3 Design

Each time ask is called, the appropriate Handler for the current key is called. This is always a subclass of `sympy.assumptions.AskHandler`. Its classmethods have the names of the classes it supports. For example, a (simplified) AskHandler for the ask 'positive' would look like this:

```
class AskPositiveHandler(CommonHandler):

    def Mul(self):
        # return True if all argument's in self.expr.args are positive
        ...

    def Add(self):
        for arg in self.expr.args:
            if not ask(arg, positive, self.assumptions):
                break
        else:
            # if all argument's are positive
            return True
        ...
```

The `.Mul()` method is called when `self.expr` is an instance of `Mul`, the `Add` method would be called when `self.expr` is an instance of `Add` and so on.

5.22.4 Extensibility

You can define new queries or support new types by subclassing `sympy.assumptions.AskHandler` and registering that handler for a particular key by calling `register_handler`:

`sympy.assumptions.ask.register_handler(key, handler)`
 Register a handler in the ask system. `key` must be a string and `handler` a class inheriting from `AskHandler`:

```
>>> from sympy.assumptions import register_handler, ask, Q
>>> from sympy.assumptions.handlers import AskHandler
>>> class MersenneHandler(AskHandler):
...     # Mersenne numbers are in the form 2**n - 1, n integer
...     @staticmethod
...     def Integer(expr, assumptions):
...         from sympy import log
...         return ask(Q.integer(log(expr + 1, 2)))
>>> register_handler('mersenne', MersenneHandler)
>>> ask(Q.mersenne(7))
True
```

You can undo this operation by calling `remove_handler`.

`sympy.assumptions.ask.remove_handler(key, handler)`
 Removes a handler from the ask system. Same syntax as `register_handler`

You can support new types¹ by adding a handler to an existing key. In the following example, we will create a new type `MyType` and extend the key `'prime'` to accept this type (and return `True`)

```
>>> from sympy.core import Basic
>>> from sympy.assumptions import register_handler
>>> from sympy.assumptions.handlers import AskHandler
>>> class MyType(Basic):
...     pass
>>> class MyAskHandler(AskHandler):
...     @staticmethod
...     def MyType(expr, assumptions):
...         return True
>>> a = MyType()
>>> register_handler('prime', MyAskHandler)
>>> ask(Q.prime(a))
True
```

5.22.5 Performance improvements

On queries that involve symbolic coefficients, logical inference is used. Work on improving satisfiable function (`sympy.logic.inference.satisfiable`) should result in notable speed improvements.

Logic inference used in one ask could be used to speed up further queries, and current system does not take advantage of this. For example, a truth maintenance system (http://en.wikipedia.org/wiki/Truth_maintenance_system) could be implemented.

5.22.6 Misc

You can find more examples in the in the form of test under directory `sympy/assumptions/tests/`

5.23 Term rewriting

Term rewriting is a very general class of functionalities which are used to convert expressions of one type in terms of expressions of different kind. For example expanding, combining and converting expressions apply to term rewriting, and also simplification routines can be included here. Currently SymPy has several functions and basic built-in methods for performing various types of rewriting.

5.23.1 Expanding

The simplest rewrite rule is expanding expressions into a `_sparse_` form. Expanding has several flavors and include expanding complex valued expressions, arithmetic expand of products and powers but also expanding functions in terms of more general functions is possible. Below are listed all currently available expand rules.

Expanding of arithmetic expressions involving products and powers:

¹ New type must inherit from `Basic`, otherwise an exception will be raised. This is a bug and should be fixed.

```
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> ((x + y)*(x - y)).expand(basic=True)
x**2 - y**2
>>> ((x + y + z)**2).expand(basic=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Arithmetic expand is done by default in `expand()` so the keyword `basic` can be omitted. However you can set `basic=False` to avoid this type of expand if you use rules described below. This gives complete control on what is done with the expression.

Another type of expand rule is expanding complex valued expressions and putting them into a normal form. For this `complex` keyword is used. Note that it will always perform arithmetic expand to obtain the desired normal form:

```
>>> (x + I*y).expand(complex=True)
re(x) + I*re(y) + I*im(x) - im(y)
```

```
>>> sin(x + I*y).expand(complex=True)
sin(re(x) - im(y))*cosh(re(y) + im(x)) + I*cos(re(x) - im(y))*sinh(re(y) + im(x))
```

Note also that the same behavior can be obtained by using `as_real_imag()` method. However it will return a tuple containing the real part in the first place and the imaginary part in the other. This can be also done in a two step process by using `collect` function:

```
>>> (x + I*y).as_real_imag()
(re(x) - im(y), re(y) + im(x))
```

```
>>> collect((x + I*y).expand(complex=True), I, evaluate=False)
{1: re(x) - im(y), I: re(y) + im(x)}
```

There is also possibility for expanding expressions in terms of expressions of different kind. This is very general type of expanding and usually you would use `rewrite()` to do specific type of rewrite:

```
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
```

5.23.2 Common Subexpression Detection and Collection

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is implemented in the `cse` function. Examples:

```
>>> from sympy import cse, sqrt, sin, pprint
>>> from sympy.abc import x

>>> pprint(cse(sqrt(sin(x))), use_unicode=True)
(
  (
    [], [
       $\sqrt{\sin(x)}$ 
    ]
  )
)

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
(
  (
    [(x_0, sin(x))], [
       $\sqrt{x_0 + 4} \cdot \sqrt{x_0 + 5}$ 
    ]
  )
)
```

(continues on next page)

(continued from previous page)

```
>>> pprint(cse(sqrt(sin(x+1) + 5 + cos(y))*sqrt(sin(x+1) + 4 + cos(y))),
...         use_unicode=True)
(
  [(x0, sin(x + 1) + cos(y))], [√x0 + 4 · √x0 + 5]
)

>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), use_unicode=True)
(
  [(x0, -y), (x1, (x + x0)*(x0 + z))], [√x1 + x1]
)
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the “optimizations” optional argument. A set of predefined basic optimizations can be applied by passing `optimizations='basic'`:

```
>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), optimizations='basic'),
...         use_unicode=True)
(
  [(x0, -(x - y)*(y - z))], [√x0 + x0]
)
```

However, these optimizations can be very slow for large expressions. Moreover, if speed is a concern, one can pass the option `order='none'`. Order of terms will then be dependent on hashing algorithm implementation, but speed will be greatly improved.

More information:

```
sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None, postprocess=None, order='canonical', ignore=())
```

Perform common subexpression elimination on an expression.

Parameters `exprs` : list of sympy expressions, or a single sympy expression

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out. The `numbered_symbols` generator is useful. The default is a stream of symbols of the form “x0”, “x1”, etc. This must be an infinite iterator.

optimizations : list of (callable, callable) pairs

The (preprocessor, postprocessor) pairs of external optimization functions. Optionally ‘basic’ can be passed for a set of predefined basic optimizations. Such ‘basic’ optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

postprocess : a function which accepts the two return values of cse and

returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda:
`lambda r, e: return reversed(r), e`

order : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. If set to ‘canonical’, arguments will be canonically ordered. If set to ‘none’, ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting `order='none'`.

ignore : iterable of Symbols

Substitutions containing any Symbol from ignore will be ignored.

Returns replacements : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

reduced_exprs : list of sympy expressions

The reduced expressions with all of the replacements above.

Examples

```
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

Note that currently, $y + z$ will not get substituted if $-y - z$ is used.

```
>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols: `>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))` `([(x0, x + 1)], [x0*y**2, 3*x0*y**2])`

5.24 Series Module

The series module implements series expansions as a function and many related functions.

5.24.1 Contents

Series Expansions

Limits

The main purpose of this module is the computation of limits.

`sympy.series.limits.limit(e, z, z0, dir='+')`

Compute the limit of $e(z)$ at the point z_0 .

z_0 can be any expression, including ∞ and $-\infty$.

For `dir="+-"` it calculates the bi-directional limit; for `dir="+"` (default) it calculates the limit from the right ($z \rightarrow z_0+$) and for `dir="-"` the limit from the left ($z \rightarrow z_0-$). For infinite z_0 (∞ or $-\infty$), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for ∞).

Notes

First we try some heuristics for easy and frequent cases like " x ", " $1/x$ ", " $x**2$ " and similar, so that it's fast. For all other cases, we use the Gruntz algorithm (see the `gruntz()` function).

Examples

```
>>> from sympy import limit, sin, Symbol, oo
>>> from sympy.abc import x
>>> limit(sin(x)/x, x, 0)
1
>>> limit(1/x, x, 0) # default dir='+'
oo
>>> limit(1/x, x, 0, dir="-")
-oo
>>> limit(1/x, x, 0, dir='+-')
Traceback (most recent call last):
...
ValueError: The limit does not exist since left hand limit = -oo and right hand
↳ limit = oo
```

```
>>> limit(1/x, x, oo)
0
```

class `sympy.series.limits.Limit`
Represents an unevaluated limit.

Examples

```
>>> from sympy import Limit, sin, Symbol
>>> from sympy.abc import x
>>> Limit(sin(x)/x, x, 0)
Limit(sin(x)/x, x, 0)
>>> Limit(1/x, x, 0, dir="-")
Limit(1/x, x, 0, dir='-')
```

doit(***hints*)
Evaluates limit

As is explained above, the workhorse for limit computations is the function `gruntz()` which implements Gruntz' algorithm for computing limits.

The Gruntz Algorithm

This section explains the basics of the algorithm used for computing limits. Most of the time the `limit()` function should just work. However it is still useful to keep in mind how it is implemented in case something does not work as expected.

First we define an ordering on functions. Suppose $f(x)$ and $g(x)$ are two real-valued functions such that $\lim_{x \rightarrow \infty} f(x) = \infty$ and similarly $\lim_{x \rightarrow \infty} g(x) = \infty$. We shall say that $f(x)$ dominates $g(x)$, written $f(x) \succ g(x)$, if for all $a, b \in \mathbb{R}_{>0}$ we have $\lim_{x \rightarrow \infty} \frac{f(x)^a}{g(x)^b} = \infty$. We also say that $f(x)$ and $g(x)$ are of the same comparability class if neither $f(x) \succ g(x)$ nor $g(x) \succ f(x)$ and shall denote it as $f(x) \asymp g(x)$.

Note that whenever $a, b \in \mathbb{R}_{>0}$ then $af(x)^b \asymp f(x)$, and we shall use this to extend the definition of $\succ$ to all functions which tend to 0 or $\pm\infty$ as $x \rightarrow \infty$. Thus we declare that $f(x) \asymp 1/f(x)$ and $f(x) \asymp -f(x)$.

It is easy to show the following examples:

- $e^x \succ x^m$
- $e^{x^2} \succ e^{mx}$
- $e^{e^x} \succ e^{x^m}$
- $x^m \asymp x^n$
- $e^{x+\frac{1}{x}} \asymp e^{x+\log x} \asymp e^x$.

From the above definition, it is possible to prove the following property:

Suppose $\omega, g_1, g_2, \dots$ are functions of x , $\lim_{x \rightarrow \infty} \omega = 0$ and $\omega \succ g_i$ for all i . Let $c_1, c_2, \dots \in \mathbb{R}$ with $c_1 < c_2 < \dots$.

Then $\lim_{x \rightarrow \infty} \sum_i g_i \omega^{c_i} = \lim_{x \rightarrow \infty} g_1 \omega^{c_1}$.

For $g_1 = g$ and ω as above we also have the following easy result:

- $\lim_{x \rightarrow \infty} g \omega^c = 0$ for $c > 0$
- $\lim_{x \rightarrow \infty} g \omega^c = \pm\infty$ for $c < 0$, where the sign is determined by the (eventual) sign of g
- $\lim_{x \rightarrow \infty} g \omega^0 = \lim_{x \rightarrow \infty} g$.

Using these results yields the following strategy for computing $\lim_{x \rightarrow \infty} f(x)$:

1. Find the set of most rapidly varying subexpressions (MRV set) of $f(x)$. That is, from the set of all subexpressions of $f(x)$, find the elements that are maximal under the relation $\succ$.
2. Choose a function ω that is in the same comparability class as the elements in the MRV set, such that $\lim_{x \rightarrow \infty} \omega = 0$.
3. Expand $f(x)$ as a series in ω in such a way that the antecedents of the above theorem are satisfied.
4. Apply the theorem and conclude the computation of $\lim_{x \rightarrow \infty} f(x)$, possibly by recursively working on $g_1(x)$.

Notes

This exposition glossed over several details. Many are described in the file `gruntz.py`, and all can be found in Gruntz' very readable thesis. The most important points that have not been explained are:

1. Given $f(x)$ and $g(x)$, how do we determine if $f(x) \succ g(x)$, $g(x) \succ f(x)$ or $g(x) \asymp f(x)$?

2. How do we find the MRV set of an expression?
3. How do we compute series expansions?
4. Why does the algorithm terminate?

If you are interested, be sure to take a look at [Gruntz Thesis](#).

Reference

`sympy.series.gruntz.gruntz(e, z, z0, dir='+')`

Compute the limit of $e(z)$ at the point z_0 using the Gruntz algorithm.

z_0 can be any expression, including ∞ and $-\infty$.

For `dir="+"` (default) it calculates the limit from the right ($z \rightarrow z_0+$) and for `dir="-"` the limit from the left ($z \rightarrow z_0-$). For infinite z_0 (∞ or $-\infty$), the `dir` argument doesn't matter.

This algorithm is fully described in the module docstring in the `gruntz.py` file. It relies heavily on the series expansion. Most frequently, `gruntz()` is only used if the faster `limit()` function (which uses heuristics) fails.

`sympy.series.gruntz.compare(a, b, x)`

Returns " $<$ " if $a < b$, " $=$ " for $a == b$, " $>$ " for $a > b$

`sympy.series.gruntz.rewrite(e, Omega, x, wsym)`

$e(x)$... the function Ω ... the mrv set $wsym$... the symbol which is going to be used for w

Returns the rewritten e in terms of w and $\log(w)$. See `test_rewrite1()` for examples and correct results.

`sympy.series.gruntz.build_expression_tree(Omega, rewrites)`

Helper function for `rewrite`.

We need to sort Ω (mrv set) so that we replace an expression before we replace any expression in terms of which it has to be rewritten:

```
e1 ---> e2 ---> e3
      \
      -> e4
```

Here we can do e_1, e_2, e_3, e_4 or e_1, e_2, e_4, e_3 . To do this we assemble the nodes into a tree, and sort them by height.

This function builds the tree, rewrites then sorts the nodes.

`sympy.series.gruntz.mrv_leadterm(e, x)`

Returns (c_0, e_0) for e .

`sympy.series.gruntz.calculate_series(e, x, logx=None)`

Calculates at least one term of the series of " e " in " x ".

This is a place that fails most often, so it is in its own function.

`sympy.series.gruntz.limitinf(e, x)`

Limit $e(x)$ for $x \rightarrow \infty$

`sympy.series.gruntz.sign(e, x)`

Returns a sign of an expression $e(x)$ for $x \rightarrow \infty$.

```
e > 0 for x sufficiently large ... 1
e == 0 for x sufficiently large ... 0
e < 0 for x sufficiently large ... -1
```

The result of this function is currently undefined if e changes sign arbitrarily often for arbitrarily large x (e.g. $\sin(x)$).

Note that this returns zero only if e is constantly zero for x sufficiently large. [If e is constant, of course, this is just the same thing as the sign of e .]

`sympy.series.gruntz.mrv(e, x)`

Returns a SubSet of most rapidly varying (mrv) subexpressions of 'e', and e rewritten in terms of these

`sympy.series.gruntz.mrv_max1(f, g, exps, x)`

Computes the maximum of two sets of expressions f and g , which are in the same comparability class, i.e. `mrv_max1()` compares (two elements of) f and g and returns the set, which is in the higher comparability class of the union of both, if they have the same order of variation. Also returns `exps`, with the appropriate substitutions made.

`sympy.series.gruntz.mrv_max3(f, expsf, g, expsg, union, expsboth, x)`

Computes the maximum of two sets of expressions f and g , which are in the same comparability class, i.e. `max()` compares (two elements of) f and g and returns either (f, expsf) [if f is larger], (g, expsg) [if g is larger] or $(\text{union}, \text{expsboth})$ [if f, g are of the same class].

class `sympy.series.gruntz.SubSet`

Stores $(\text{expr}, \text{dummy})$ pairs, and how to rewrite expr -s.

The gruntz algorithm needs to rewrite certain expressions in term of a new variable w . We cannot use `subs`, because it is just too smart for us. For example:

```
> Omega=[exp(exp(_p - exp(-_p))/(1 - 1/_p)), exp(exp(_p))]
> O2=[exp(-exp(_p) + exp(-exp(-_p))*exp(_p)/(1 - 1/_p))/_w, 1/_w]
> e = exp(exp(_p - exp(-_p))/(1 - 1/_p)) - exp(exp(_p))
> e.subs(Omega[0],O2[0]).subs(Omega[1],O2[1])
-1/w + exp(exp(p)*exp(-exp(-p))/(1 - 1/p))
```

is really not what we want!

So we do it the hard way and keep track of all the things we potentially want to substitute by dummy variables. Consider the expression:

```
exp(x - exp(-x)) + exp(x) + x.
```

The mrv set is $\{\exp(x), \exp(-x), \exp(x - \exp(-x))\}$. We introduce corresponding dummy variables $d1, d2, d3$ and rewrite:

```
d3 + d1 + x.
```

This class first of all keeps track of the mapping $\text{expr} \rightarrow \text{variable}$, i.e. will at this stage be a dictionary:

```
{exp(x): d1, exp(-x): d2, exp(x - exp(-x)): d3}.
```

[It turns out to be more convenient this way round.] But sometimes expressions in the mrv set have other expressions from the mrv set as subexpressions, and we need to keep track of that as well. In this case, $d3$ is really $\exp(x - d2)$, so rewrites at this stage is:

```
{d3: exp(x-d2)}.
```

The function `rewrite` uses all this information to correctly rewrite our expression in terms of w . In this case w can be chosen to be $\exp(-x)$, i.e. $d2$. The correct rewriting then is:

```
exp(-w)/w + 1/w + x.
```

copy()

Create a shallow copy of `SubsSet`

do_subs(e)

Substitute the variables with expressions

meets(s2)

Tell whether or not self and `s2` have non-empty intersection

union(s2, exps=None)

Compute the union of self and `s2`, adjusting `exps`

More Intuitive Series Expansion

This is achieved by creating a wrapper around `Basic.series()`. This allows for the use of `series(x*cos(x),x)`, which is possibly more intuitive than `(x*cos(x)).series(x)`.

Examples

```
>>> from sympy import Symbol, cos, series
>>> x = Symbol('x')
>>> series(cos(x),x)
1 - x**2/2 + x**4/24 + O(x**6)
```

Reference

`sympy.series.series.series(expr, x=None, x0=0, n=6, dir='+')`
Series expansion of `expr` around point $x = x_0$.

See the doctring of `Expr.series()` for complete details of this wrapper.

Order Terms

This module also implements automatic keeping track of the order of your expansion.

Examples

```
>>> from sympy import Symbol, Order
>>> x = Symbol('x')
>>> Order(x) + x**2
O(x)
>>> Order(x) + 1
1 + O(x)
```

Reference

`class sympy.series.order.Order`

Represents the limiting behavior of some function

The order of a function characterizes the function based on the limiting behavior of the function as it goes to some limit. Only taking the limit point to be a number is currently supported. This is expressed in big O notation [R457] (page 1868).

The formal definition for the order of a function $g(x)$ about a point a is such that $g(x) = O(f(x))$ as $x \rightarrow a$ if and only if for any $\delta > 0$ there exists a $M > 0$ such that $|g(x)| \leq M|f(x)|$ for $|x - a| < \delta$. This is equivalent to $\lim_{x \rightarrow a} \sup |g(x)/f(x)| < \infty$.

Let's illustrate it on the following example by taking the expansion of $\sin(x)$ about 0:

$$\sin(x) = x - x^3/3! + O(x^5)$$

where in this case $O(x^5) = x^5/5! - x^7/7! + \dots$. By the definition of O , for any $\delta > 0$ there is an M such that:

$$|x^5/5! - x^7/7! + \dots| \leq M|x^5| \text{ for } |x| < \delta$$

or by the alternate definition:

$$\lim_{x \rightarrow 0} |(x^5/5! - x^7/7! + \dots)/x^5| < \infty$$

which surely is true, because

$$\lim_{x \rightarrow 0} |(x^5/5! - x^7/7! + \dots)/x^5| = 1/5!$$

As it is usually used, the order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example, $O(x^3)$ corresponds to any terms proportional to $x^3, x^4, \dots$ and any higher power. For a polynomial, this leaves terms proportional to x^2, x and constants.

Notes

In $O(f(x), x)$ the expression $f(x)$ is assumed to have a leading term. $O(f(x), x)$ is automatically transformed to $O(f(x).as\_leading\_term(x), x)$.

$O(\text{expr} * f(x), x)$ is $O(f(x), x)$

$O(\text{expr}, x)$ is $O(1)$

$O(0, x)$ is 0.

Multivariate O is also supported:

$O(f(x, y), x, y)$ is transformed to $O(f(x, y).as\_leading\_term(x, y).as\_leading\_term(y), x, y)$

In the multivariate case, it is assumed the limits w.r.t. the various symbols commute.

If no symbols are passed then all symbols in the expression are used and the limit point is assumed to be zero.

References

[R457] (page 1868)

Examples

```
>>> from sympy import O, oo, cos, pi
>>> from sympy.abc import x, y
```

```
>>> O(x + x**2)
O(x)
>>> O(x + x**2, (x, 0))
O(x)
>>> O(x + x**2, (x, oo))
O(x**2, (x, oo))
```

```
>>> O(1 + x*y)
O(1, x, y)
>>> O(1 + x*y, (x, 0), (y, 0))
O(1, x, y)
>>> O(1 + x*y, (x, oo), (y, oo))
O(x*y, (x, oo), (y, oo))
```

```
>>> O(1) in O(1, x)
True
>>> O(1, x) in O(1)
False
>>> O(x) in O(1, x)
True
>>> O(x**2) in O(x)
True
```

```
>>> O(x)*x
O(x**2)
>>> O(x) - O(x)
O(x)
>>> O(cos(x))
O(1)
>>> O(cos(x), (x, pi/2))
O(x - pi/2, (x, pi/2))
```

contains(expr)

Return True if expr belongs to $\text{Order}(\text{self.expr}, * \text{self.variables})$. Return False if self belongs to expr. Return None if the inclusion relation cannot be determined (e.g. when self and expr have different symbols).

Series Acceleration

TODO

Reference

`sympy.series.acceleration.richardson(A, k, n, N)`

Calculate an approximation for $\lim_{k \rightarrow \infty} A(k)$ using Richardson extrapolation with the terms $A(n)$, $A(n+1)$, ..., $A(n+N+1)$. Choosing $N \sim 2^n$ often gives good results.

A simple example is to calculate $\exp(1)$ using the limit definition. This limit converges slowly; $n = 100$ only produces two accurate digits:

```
>>> from sympy.abc import n
>>> e = (1 + 1/n)**n
>>> print(round(e.subs(n, 100).evalf(), 10))
2.7048138294
```

Richardson extrapolation with 11 appropriately chosen terms gives a value that is accurate to the indicated precision:

```
>>> from sympy import E
>>> from sympy.series.acceleration import richardson
>>> print(round(richardson(e, n, 10, 20).evalf(), 10))
2.7182818285
>>> print(round(E.evalf(), 10))
2.7182818285
```

Another useful application is to speed up convergence of series. Computing 100 terms of the zeta(2) series $1/k^{**2}$ yields only two accurate digits:

```
>>> from sympy.abc import k, n
>>> from sympy import Sum
>>> A = Sum(k**(-2), (k, 1, n))
>>> print(round(A.subs(n, 100).evalf(), 10))
1.6349839002
```

Richardson extrapolation performs much better:

```
>>> from sympy import pi
>>> print(round(richardson(A, n, 10, 20).evalf(), 10))
1.6449340668
>>> print(round(((pi**2)/6).evalf(), 10))      # Exact value
1.6449340668
```

`sympy.series.acceleration.shanks(A, k, n, m=1)`

Calculate an approximation for $\lim_{k \rightarrow \infty} A(k)$ using the n -term Shanks transformation $S(A)(n)$. With $m > 1$, calculate the m -fold recursive Shanks transformation $S(S(\dots S(A)\dots))(n)$.

The Shanks transformation is useful for summing Taylor series that converge slowly near a pole or singularity, e.g. for $\log(2)$:

```
>>> from sympy.abc import k, n
>>> from sympy import Sum, Integer
>>> from sympy.series.acceleration import shanks
>>> A = Sum(Integer(-1)**(k+1) / k, (k, 1, n))
>>> print(round(A.subs(n, 100).doit().evalf(), 10))
0.6881721793
>>> print(round(shanks(A, n, 25).evalf(), 10))
0.6931396564
>>> print(round(shanks(A, n, 25, 5).evalf(), 10))
0.6931471806
```

The correct value is 0.6931471805599453094172321215.

Residues

TODO

Reference

`sympy.series.residues.residue(expr, x, x0)`

Finds the residue of `expr` at the point `x=x0`.

The residue is defined as the coefficient of $1/(x-x_0)$ in the power series expansion about $x=x_0$.

References

1. http://en.wikipedia.org/wiki/Residue_theorem

Examples

```
>>> from sympy import Symbol, residue, sin
>>> x = Symbol("x")
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

This function is essential for the Residue Theorem [1].

Sequences

A sequence is a finite or infinite lazily evaluated list.

`sympy.series.sequences.sequence(seq, limits=None)`

Returns appropriate sequence object.

If `seq` is a sympy sequence, returns `SeqPer` (page 1117) object otherwise returns `SeqFormula` (page 1116) object.

See also:

`sympy.series.sequences.SeqPer` (page 1117), `sympy.series.sequences.SeqFormula` (page 1116)

Examples

```
>>> from sympy import sequence, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> sequence(n**2, (n, 0, 5))
SeqFormula(n**2, (n, 0, 5))
>>> sequence((1, 2, 3), (n, 0, 5))
SeqPer((1, 2, 3), (n, 0, 5))
```


Sequences Base

class `sympy.series.sequences.SeqBase`

Base class for sequences

coeff(pt)

Returns the coefficient at point pt

coeff_mul(other)

Should be used when other is not a sequence. Should be defined to define custom behaviour.

Notes

'*' defines multiplication of sequences with sequences only.

Examples

```
>>> from sympy import S, oo, SeqFormula
>>> from sympy.abc import n
>>> SeqFormula(n**2).coeff_mul(2)
SeqFormula(2*n**2, (n, 0, oo))
```

find_linear_recurrence(n, d=None, gfvar=None)

Finds the shortest linear recurrence that satisfies the first n terms of sequence of order $\leq n/2$ if possible. If d is specified, find shortest linear recurrence of order $\leq \min(d, n/2)$ if possible. Returns list of coefficients $[b(1), b(2), \dots]$ corresponding to the recurrence relation $x(n) = b(1)*x(n-1) + b(2)*x(n-2) + \dots$. Returns $[]$ if no recurrence is found. If gfvar is specified, also returns ordinary generating function as a function of gfvar.

Examples

```
>>> from sympy import sequence, sqrt, oo, lucas
>>> from sympy.abc import n, x, y
>>> sequence(n**2).find_linear_recurrence(10, 2)
[]
>>> sequence(n**2).find_linear_recurrence(10)
[3, -3, 1]
>>> sequence(2**n).find_linear_recurrence(10)
[2]
>>> sequence(23*n**4+91*n**2).find_linear_recurrence(10)
[5, -10, 10, -5, 1]
>>> sequence(sqrt(5)*(((1 + sqrt(5))/2)**n - (-(1 + sqrt(5))/2)**(-n))/5).
    find_linear_recurrence(10)
[1, 1]
>>> sequence(x+y*(-2)**(-n), (n, 0, oo)).find_linear_recurrence(30)
[1/2, 1/2]
>>> sequence(3*5**n + 12).find_linear_recurrence(20,gfvar=x)
([6, -5], 3*(-21*x + 5)/((x - 1)*(5*x - 1)))
>>> sequence(lucas(n)).find_linear_recurrence(15,gfvar=x)
([1, 1], (x - 2)/(x**2 + x - 1))
```

free_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> from sympy import SeqFormula
>>> from sympy.abc import n, m
>>> SeqFormula(m*n**2, (n, 0, 5)).free_symbols
{m}
```

gen

Returns the generator for the sequence

interval

The interval on which the sequence is defined

length

Length of the sequence

start

The starting point of the sequence. This point is included

stop

The ending point of the sequence. This point is included

variables

Returns a tuple of variables that are bounded

Elementary Sequences**class** sympy.series.sequences.**SeqFormula**

Represents sequence based on a formula.

Elements are generated using a formula.

See also:

[sympy.series.sequences.SeqPer](#) (page 1117)

Examples

```
>>> from sympy import SeqFormula, oo, Symbol
>>> n = Symbol('n')
>>> s = SeqFormula(n**2, (n, 0, 5))
>>> s.formula
n**2
```

For value at a particular point

```
>>> s.coeff(3)
9
```

supports slicing

```
>>> s[:]
[0, 1, 4, 9, 16, 25]
```

iterable

```
>>> list(s)
[0, 1, 4, 9, 16, 25]
```

sequence starts from negative infinity

```
>>> SeqFormula(n**2, (-oo, 0))[0:6]
[0, 1, 4, 9, 16, 25]
```

coeff_mul(coeff)
See docstring of SeqBase.coeff_mul

class sympy.series.sequences.**SeqPer**

Represents a periodic sequence.

The elements are repeated after a given period.

See also:

[sympy.series.sequences.SeqFormula](#) (page 1116)

Examples

```
>>> from sympy import SeqPer, oo
>>> from sympy.abc import k
```

```
>>> s = SeqPer((1, 2, 3), (0, 5))
>>> s.periodical
(1, 2, 3)
>>> s.period
3
```

For value at a particular point

```
>>> s.coeff(3)
1
```

supports slicing

```
>>> s[:]
[1, 2, 3, 1, 2, 3]
```

iterable

```
>>> list(s)
[1, 2, 3, 1, 2, 3]
```

sequence starts from negative infinity

```
>>> SeqPer((1, 2, 3), (-oo, 0))[0:6]
[1, 2, 3, 1, 2, 3]
```

Periodic formulas

```
>>> SeqPer((k, k**2, k**3), (k, 0, oo))[0:6]
[0, 1, 8, 3, 16, 125]
```

coeff_mul(coeff)
See docstring of SeqBase.coeff_mul

Singleton Sequences

class sympy.series.sequences.EmptySequence

Represents an empty sequence.

The empty sequence is available as a singleton as S.EmptySequence.

Examples

```
>>> from sympy import S, SeqPer, oo
>>> from sympy.abc import x
>>> S.EmptySequence
EmptySequence()
>>> SeqPer((1, 2), (x, 0, 10)) + S.EmptySequence
SeqPer((1, 2), (x, 0, 10))
>>> SeqPer((1, 2)) * S.EmptySequence
EmptySequence()
>>> S.EmptySequence.coeff_mul(-1)
EmptySequence()
```

coeff_mul(coeff)
See docstring of SeqBase.coeff_mul

Compound Sequences

class sympy.series.sequences.SeqAdd

Represents term-wise addition of sequences.

Rules:

- The interval on which sequence is defined is the intersection of respective intervals of sequences.
- Anything + [EmptySequence](#) (page 1118) remains unchanged.
- Other rules are defined in `_add` methods of sequence classes.

See also:

[sympy.series.sequences.SeqMul](#) (page 1119)

Examples

```
>>> from sympy import S, oo, SeqAdd, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), S.EmptySequence)
SeqPer((1, 2), (n, 0, oo))
```

(continues on next page)

(continued from previous page)

```
>>> SeqAdd(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence()
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2, (n, 0, oo)))
SeqAdd(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqAdd(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**3 + n**2, (n, 0, oo))
```

static reduce(args)

Simplify [SeqAdd](#) (page 1118) using known rules.

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from `Union.reduce`

class sympy.series.sequences.SeqMul

Represents term-wise multiplication of sequences.

Handles multiplication of sequences only. For multiplication with other objects see [SeqBase.coeff_mul\(\)](#) (page 1115).

Rules:

- The interval on which sequence is defined is the intersection of respective intervals of sequences.
- Anything * [EmptySequence](#) (page 1118) returns [EmptySequence](#) (page 1118).
- Other rules are defined in `_mul` methods of sequence classes.

See also:

[sympy.series.sequences.SeqAdd](#) (page 1118)

Examples

```
>>> from sympy import S, oo, SeqMul, SeqPer, SeqFormula
>>> from sympy.abc import n
>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), S.EmptySequence)
EmptySequence()
>>> SeqMul(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence()
>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2))
SeqMul(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqMul(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**5, (n, 0, oo))
```

static reduce(args)

Simplify a [SeqMul](#) (page 1119) using known rules.

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from `Union.reduce`

Fourier Series

Provides methods to compute Fourier series.

class `sympy.series.fourier.FourierSeries`

Represents Fourier sine/cosine series.

This class only represents a fourier series. No computation is performed.

For how to compute Fourier series, see the `fourier_series()` (page 1122) docstring.

See also:

`sympy.series.fourier.fourier_series` (page 1122)

scale(s)

Scale the function by a term independent of x .

$f(x) \rightarrow s * f(x)$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
```

scalex(s)

Scale x by a term independent of x .

$f(x) \rightarrow f(s*x)$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

shift(s)

Shift the function by a term independent of x .

$f(x) \rightarrow f(x) + s$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
```

shiftx(s)

Shift x by a term independent of x.

$f(x) \rightarrow f(x + s)$

This is fast, if Fourier series of $f(x)$ is already computed.

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

sigma_approximation(n=3)

Return σ -approximation of Fourier series with respect to order n.

Sigma approximation adjusts a Fourier summation to eliminate the Gibbs phenomenon which would otherwise occur at discontinuities. A sigma-approximated summation for a Fourier series of a T-periodical function can be written as

$$s(\theta) = \frac{1}{2}a_0 + \sum_{k=1}^{m-1} \text{sinc}\left(\frac{k}{m}\right) \cdot \left[a_k \cos\left(\frac{2\pi k}{T}\theta\right) + b_k \sin\left(\frac{2\pi k}{T}\theta\right) \right],$$

where $a_0, a_k, b_k, k = 1, \dots, m-1$ are standard Fourier series coefficients and $\text{sinc}\left(\frac{k}{m}\right)$ is a Lanczos σ factor (expressed in terms of normalized sinc function).

Parameters n : int

Highest order of the terms taken into account in approximation.

Returns Expr

Sigma approximation of function expanded into Fourier series.

See also:

[sympy.series.fourier.FourierSeries.truncate](#) (page 1122)

Notes

The behaviour of [sigma_approximation\(\)](#) (page 1121) is different from [truncate\(\)](#) (page 1122) - it takes all nonzero terms of degree smaller than n, rather than first n nonzero ones.

References

[R452] (page 1868), [R453] (page 1868)

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.sigma_approximation(4)
2*sin(x)*sinc(pi/4) - 2*sin(2*x)/pi + 2*sin(3*x)*sinc(3*pi/4)/3
```

truncate(n=3)

Return the first n nonzero terms of the series.

If n is None return an iterator.

Parameters n : int or None

Amount of non-zero terms in approximation or None.

Returns Expr or iterator

Approximation of function expanded into Fourier series.

See also:

[sympy.series.fourier.FourierSeries.sigma_approximation](#) (page 1121)

Examples

```
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.truncate(4)
2*sin(x) - sin(2*x) + 2*sin(3*x)/3 - sin(4*x)/2
```

[sympy.series.fourier.fourier_series](#)(f, limits=None)

Computes Fourier sine/cosine series expansion.

Returns a [FourierSeries](#) (page 1120) object.

See also:

[sympy.series.fourier.FourierSeries](#) (page 1120)

Notes

Computing Fourier series can be slow due to the integration required in computing a_n .

It is faster to compute Fourier series of a function by using shifting and scaling on an already computed Fourier series rather than computing again.

e.g. If the Fourier series of x^2 is known the Fourier series of $x^2 - 1$ can be found by shifting by -1.

References

[R454] (page 1868)

Examples

```
>>> from sympy import fourier_series, pi, cos
>>> from sympy.abc import x
```

```
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.truncate(n=3)
-4*cos(x) + cos(2*x) + pi**2/3
```

Shifting

```
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

Scaling

```
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

Formal Power Series

Methods for computing and manipulating Formal Power Series.

class `sympy.series.formal.FormalPowerSeries`

Represents Formal Power Series of a function.

No computation is performed. This class should only to be used to represent a series.
No checks are performed.

For computing a series use `fps()` (page 1124).

See also:

`sympy.series.formal.fps` (page 1124)

infinite

Returns an infinite representation of the series

integrate(`x=None`, `**kwargs`)

Integrate Formal Power Series.

Examples

```
>>> from sympy import fps, sin, integrate
>>> from sympy.abc import x
>>> f = fps(sin(x))
```

(continues on next page)

(continued from previous page)

```
>>> f.integrate(x).truncate()
-1 + x**2/2 - x**4/24 + O(x**6)
>>> integrate(f, (x, 0, 1))
-cos(1) + 1
```

polynomial(n=6)

Truncated series as polynomial.

Returns series sexpansion of f upto order $O(x^n)$ as a polynomial(without 0 term).**truncate**(n=6)

Truncated series.

Returns truncated series expansion of f upto order $O(x^n)$.If n is None, returns an infinite iterator.**sympy.series.formal.fps**(f , $x=None$, $x0=0$, $dir=1$, $hyper=True$, $order=4$, $rational=True$, $full=False$)Generates Formal Power Series of f .Returns the formal series expansion of f around $x = x0$ with respect to x in the form of a `FormalPowerSeries` object.

Formal Power Series is represented using an explicit formula computed using different algorithms.

See [compute_fps\(\)](#) (page 1125) for the more details regarding the computation of formula.**Parameters** x : Symbol, optionalIf x is None and f is univariate, the univariate symbols will be supplied, otherwise an error will be raised. **$x0$** : number, optional

Point to perform series expansion about. Default is 0.

 dir : {1, -1, '+', '-'}, optionalIf dir is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1. **$hyper$** : {True, False}, optionalSet $hyper$ to False to skip the hypergeometric algorithm. By default it is set to False. **$order$** : int, optionalOrder of the derivative of f , Default is 4. **$rational$** : {True, False}, optionalSet $rational$ to False to skip rational algorithm. By default it is set to True. **$full$** : {True, False}, optionalSet $full$ to True to increase the range of rational algorithm. See [rational_algorithm\(\)](#) (page 1126) for details. By default it is set to False.

See also:

`sympy.series.formal.FormalPowerSeries` (page 1123), `sympy.series.formal.compute_fps` (page 1125)

Examples

```
>>> from sympy import fps, 0, ln, atan
>>> from sympy.abc import x
```

Rational Functions

```
>>> fps(ln(1 + x)).truncate()
x - x**2/2 + x**3/3 - x**4/4 + x**5/5 + O(x**6)
```

```
>>> fps(atan(x), full=True).truncate()
x - x**3/3 + x**5/5 + O(x**6)
```

`sympy.series.formal.compute_fps(f, x, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)`

Computes the formula for Formal Power Series of a function.

Tries to compute the formula by applying the following techniques (in order):

- `rational_algorithm`
- Hypergeometric algorithm

Parameters **x** : Symbol

x0 : number, optional

Point to perform series expansion about. Default is 0.

dir : {1, -1, '+', '-'}, optional

If `dir` is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.

hyper : {True, False}, optional

Set `hyper` to False to skip the hypergeometric algorithm. By default it is set to False.

order : int, optional

Order of the derivative of `f`, Default is 4.

rational : {True, False}, optional

Set `rational` to False to skip rational algorithm. By default it is set to True.

full : {True, False}, optional

Set `full` to True to increase the range of rational algorithm. See `rational_algorithm()` (page 1126) for details. By default it is set to False.

Returns **ak** : sequence

Sequence of coefficients.

xk : sequence
Sequence of powers of x.
ind : Expr
Independent terms.
mul : Pow
Common terms.

See also:

[sympy.series.formal.rational_algorithm](#) (page 1126), [sympy.series.formal.hyper_algorithm](#) (page 1129)

Rational Algorithm

`sympy.series.formal.rational_independent`(terms, x)
Returns a list of all the rationally independent terms.

Examples

```
>>> from sympy import sin, cos
>>> from sympy.series.formal import rational_independent
>>> from sympy.abc import x
```

```
>>> rational_independent([cos(x), sin(x)], x)
[cos(x), sin(x)]
>>> rational_independent([x**2, sin(x), x*sin(x), x**3], x)
[x**3 + x**2, x*sin(x) + sin(x)]
```

`sympy.series.formal.rational_algorithm`(f, x, k, order=4, full=False)
Rational algorithm for computing formula of coefficients of Formal Power Series of a function.

Applicable when $f(x)$ or some derivative of $f(x)$ is a rational function in x .

`rational_algorithm()` (page 1126) uses `apart()` function for partial fraction decomposition. `apart()` by default uses 'undetermined coefficients method'. By setting `full=True`, 'Bronstein's algorithm' can be used instead.

Looks for derivative of a function up to 4'th order (by default). This can be overridden using `order` option.

Returns formula : Expr
ind : Expr
Independent terms.
order : int

See also:

[sympy.polys.partfrac.apart](#) (page 876)

Notes

By setting `full=True`, range of admissible functions to be solved using `rational_algorithm` can be increased. This option should be used carefully as it can significantly slow down the computation as `doit` is performed on the `RootSum` object returned by the `apart` function. Use `full=False` whenever possible.

References

[R448] (page 1868), [R449] (page 1868)

Examples

```
>>> from sympy import log, atan, I
>>> from sympy.series.formal import rational_algorithm as ra
>>> from sympy.abc import x, k
```

```
>>> ra(1 / (1 - x), x, k)
(1, 0, 0)
>>> ra(log(1 + x), x, k)
((-1)**(-k)/k, 0, 1)
```

```
>>> ra(atan(x), x, k, full=True)
((-I*(-I)**(-k)/2 + I*I**(-k)/2)/k, 0, 1)
```

Hypergeometric Algorithm

`sympy.series.formal.simpleDE(f, x, g, order=4)`

Generates simple DE.

DE is of the form

$$f^k(x) + \sum_{j=0}^{k-1} A_j f^j(x) = 0$$

where A_j should be rational function in x .

Generates DE's upto order 4 (default). DE's can also have free parameters.

By increasing order, higher order DE's can be found.

Yields a tuple of (DE, order).

`sympy.series.formal.exp_re(DE, r, k)`

Converts a DE with constant coefficients (explike) into a RE.

Performs the substitution:

$$f^j(x) \rightarrow r(k+j)$$

Normalises the terms so that lowest order of a term is always $r(k)$.

See also:

`sympy.series.formal.hyper_re` (page 1128)

Examples

```
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import exp_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')
```

```
>>> exp_re(-f(x) + Derivative(f(x)), r, k)
-r(k) + r(k + 1)
>>> exp_re(Derivative(f(x), x) + Derivative(f(x), (x, 2)), r, k)
r(k) + r(k + 1)
```

`sympy.series.formal.hyper_re(DE, r, k)`

Converts a DE into a RE.

Performs the substitution:

$$x^l f^j(x) \rightarrow (k + 1 - l)_j \cdot a_{k+j-l}$$

Normalises the terms so that lowest order of a term is always $r(k)$.

See also:

`sympy.series.formal.exp_re` (page 1127)

Examples

```
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import hyper_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')
```

```
>>> hyper_re(-f(x) + Derivative(f(x)), r, k)
(k + 1)*r(k + 1) - r(k)
>>> hyper_re(-x*f(x) + Derivative(f(x), (x, 2)), r, k)
(k + 2)*(k + 3)*r(k + 3) - r(k)
```

`sympy.series.formal.resolve_hypergeometric(f, x, P, Q, k, m)`

Solves RE of hypergeometric type.

Attempts to solve RE of the form

$$Q(k)*a(k + m) - P(k)*a(k)$$

Transformations that preserve Hypergeometric type:

1. $x**n*f(x)$: $b(k + m) = R(k - n)*b(k)$
2. $f(A*x)$: $b(k + m) = A**m*R(k)*b(k)$
3. $f(x**n)$: $b(k + n*m) = R(k/n)*b(k)$
4. $f(x**(1/m))$: $b(k + 1) = R(k*m)*b(k)$
5. $f'(x)$: $b(k + m) = ((k + m + 1)/(k + 1))*R(k + 1)*b(k)$

Some of these transformations have been used to solve the RE.

Returns formula : Expr

ind : Expr

Independent terms.

order : int

References

[R450] (page 1868), [R451] (page 1868)

Examples

```
>>> from sympy import exp, ln, S
>>> from sympy.series.formal import rsolve_hypergeometric as rh
>>> from sympy.abc import x, k
```

```
>>> rh(exp(x), x, -S.One, (k + 1), k, 1)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> rh(ln(1 + x), x, k**2, k*(k + 1), k, 1)
(Piecewise((((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

`sympy.series.formal.solve_de(f, x, DE, order, g, k)`

Solves the DE.

Tries to solve DE by either converting into a RE containing two terms or converting into a DE having constant coefficients.

Returns formula : Expr

ind : Expr

Independent terms.

order : int

Examples

```
>>> from sympy import Derivative as D
>>> from sympy import exp, ln
>>> from sympy.series.formal import solve_de
>>> from sympy.abc import x, k, f
```

```
>>> solve_de(exp(x), x, D(f(x), x) - f(x), 1, f, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> solve_de(ln(1 + x), x, (x + 1)*D(f(x), x, 2) + D(f(x)), 2, f, k)
(Piecewise(((((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

`sympy.series.formal.hyper_algorithm(f, x, k, order=4)`

Hypergeometric algorithm for computing Formal Power Series.

Steps:

- Generates DE

- Convert the DE into RE
- Solves the RE

See also:

`sympy.series.formal.simpleDE` (page 1127), `sympy.series.formal.solve_de` (page 1129)

Examples

```
>>> from sympy import exp, ln
>>> from sympy.series.formal import hyper_algorithm
```

```
>>> from sympy.abc import x, k
```

```
>>> hyper_algorithm(exp(x), x, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)
```

```
>>> hyper_algorithm(ln(1 + x), x, k)
(Piecewise((((1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

Limits of Sequences

Provides methods to compute limit of terms having sequences at infinity.

`sympy.series.limitseq.difference_delta(expr, n=None, step=1)`

Difference Operator.

Discrete analog of differential operator. Given a sequence $x[n]$, returns the sequence $x[n + \text{step}] - x[n]$.

References

[R455] (page 1868)

Examples

```
>>> from sympy import difference_delta as dd
>>> from sympy.abc import n
>>> dd(n*(n + 1), n)
2*n + 2
>>> dd(n*(n + 1), n, 2)
4*n + 6
```

`sympy.series.limitseq.dominant(expr, n)`

Finds the dominant term in a sum, that is a term that dominates every other term.

If $\lim(a/b, n, \infty)$ is ∞ then a dominates b . If $\lim(a/b, n, \infty)$ is 0 then b dominates a . Otherwise, a and b are comparable.

If there is no unique dominant term, then returns `None`.

See also:

`sympy.series.limitseq.dominant` (page 1130)

Examples

```
>>> from sympy import Sum
>>> from sympy.series.limitseq import dominant
>>> from sympy.abc import n, k
>>> dominant(5*n**3 + 4*n**2 + n + 1, n)
5*n**3
>>> dominant(2**n + Sum(k, (k, 0, n)), n)
2**n
```

`sympy.series.limitseq.limit_seq(expr, n=None, trials=5)`

Finds limits of terms having sequences at infinity.

Parameters `expr` : Expr

SymPy expression for the n-th term of the sequence

`n` : Symbol

The index of the sequence, an integer that tends to positive infinity.

trials: int, optional

The algorithm is highly recursive. `trials` is a safeguard from infinite recursion in case the limit is not easily computed by the algorithm. Try increasing `trials` if the algorithm returns `None`.

See also:

`sympy.series.limitseq.dominant` (page 1130)

References

[R456] (page 1868)

Examples

```
>>> from sympy import limit_seq, Sum, binomial
>>> from sympy.abc import n, k, m
>>> limit_seq((5*n**3 + 3*n**2 + 4) / (3*n**3 + 4*n - 5), n)
5/3
>>> limit_seq(binomial(2*n, n) / Sum(binomial(2*k, k), (k, 1, n)), n)
3/4
>>> limit_seq(Sum(k**2 * Sum(2**m/m, (m, 1, k)), (k, 1, n)) / (2**n*n), n)
4
```

Admissible Terms

The algorithm is designed for sequences built from rational functions, indefinite sums, and indefinite products over an indeterminate `n`. Terms of alternating sign are also allowed, but more complex oscillatory behavior is not supported.

5.25 Sets

5.25.1 Set

class `sympy.sets.sets.Set`

The base class for any kind of set.

This is not meant to be used directly as a container of items. It does not behave like the builtin set; see [FiniteSet](#) (page 1141) for that.

Real intervals are represented by the [Interval](#) (page 1138) class and unions of sets by the [Union](#) (page 1141) class. The empty set is represented by the [EmptySet](#) (page 1145) class and available as a singleton as `S.EmptySet`.

Attributes

<code>is_Complement</code>	
<code>is_EmptySet</code>	
<code>is_Intersection</code>	
<code>is_UniversalSet</code>	

boundary

The boundary or frontier of a set

A point x is on the boundary of a set S if

1. x is in the closure of S . I.e. Every neighborhood of x contains a point in S .
2. x is not in the interior of S . I.e. There does not exist an open set centered on x contained entirely within S .

There are the points on the outer rim of S . If S is open then these points need not actually be contained within S .

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).boundary
{0, 1}
>>> Interval(0, 1, True, False).boundary
{0, 1}
```

closure

Property method which returns the closure of a set. The closure is defined as the union of the set itself and its boundary.

Examples

```
>>> from sympy import S, Interval
>>> S.Reals.closure
Reals
>>> Interval(0, 1).closure
Interval(0, 1)
```

complement(universe)

The complement of 'self' w.r.t the given universe.

Examples

```
>>> from sympy import Interval, S
>>> Interval(0, 1).complement(S.Reals)
Union(Interval.open(-oo, 0), Interval.open(1, oo))
```

```
>>> Interval(0, 1).complement(S.UniversalSet)
UniversalSet() \ Interval(0, 1)
```

contains(other)

Returns True if 'other' is contained in 'self' as an element.

As a shortcut it is possible to use the 'in' operator:

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).contains(0.5)
True
>>> 0.5 in Interval(0, 1)
True
```

inf

The infimum of 'self'

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

interior

Property method which returns the interior of a set. The interior of a set S consists all points of S that do not belong to the boundary of S.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).interior
Interval.open(0, 1)
>>> Interval(0, 1).boundary.interior
EmptySet()
```

intersect(other)

Returns the intersection of 'self' and 'other'.

```
>>> from sympy import Interval
```

```
>>> Interval(1, 3).intersect(Interval(1, 2))
Interval(1, 2)
```

```
>>> from sympy import imageset, Lambda, symbols, S
>>> n, m = symbols('n m')
>>> a = imageset(Lambda(n, 2*n), S.Integers)
>>> a.intersect(imageset(Lambda(m, 2*m + 1), S.Integers))
EmptySet()
```

intersection(other)

Alias for `intersect()` (page 1134)

is_closed

A property method to check whether a set is closed. A set is closed if it's complement is an open set.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).is_closed
True
```

is_disjoint(other)

Returns True if 'self' and 'other' are disjoint

References

[R458] (page 1868)

Examples

```
>>> from sympy import Interval
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

is_open

Property method to check whether a set is open. A set is open if and only if it has an empty intersection with its boundary.

Examples

```
>>> from sympy import S
>>> S.Reals.is_open
True
```

is_proper_subset(other)

Returns True if 'self' is a proper subset of 'other'.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

is_proper_superset(other)

Returns True if 'self' is a proper superset of 'other'.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

is_subset(other)

Returns True if 'self' is a subset of 'other'.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

is_superset(other)

Returns True if 'self' is a superset of 'other'.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superset(Interval(0, 1, left_open=True))
True
```

isdisjoint(other)

Alias for `is_disjoint()` (page 1134)

issubset(other)

Alias for `is_subset()` (page 1135)

issuperset(other)

Alias for `is_superset()` (page 1135)

measure

The (Lebesgue) measure of 'self'

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

powerset()

Find the Power set of 'self'.

References

[R459] (page 1868)

Examples

```
>>> from sympy import FiniteSet, EmptySet
>>> A = EmptySet()
>>> A.powerset()
{EmptySet()}
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet())
True
```

sup

The supremum of 'self'

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

symmetric_difference(other)

Returns symmetric difference of *self* and *other*.

References

[R460] (page 1868)

Examples

```
>>> from sympy import Interval, S
>>> Interval(1, 3).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(3, oo))
>>> Interval(1, 10).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(10, oo))
```

```
>>> from sympy import S, EmptySet
>>> S.Reals.symmetric_difference(EmptySet())
Reals
```

union(other)

Returns the union of ‘self’ and ‘other’.

Examples

As a shortcut it is possible to use the ‘+’ operator:

```
>>> from sympy import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(0, 1) + Interval(2, 3)
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
Union(Interval.Lopen(1, 2), {3})
```

Similarly it is possible to use the ‘-’ operator for set differences:

```
>>> Interval(0, 2) - Interval(0, 1)
Interval.Lopen(1, 2)
>>> Interval(1, 3) - FiniteSet(2)
Union(Interval.Ropen(1, 2), Interval.Lopen(2, 3))
```

sympy.sets.sets.imageset(*args)

Return an image of the set under transformation f .

If this function can’t compute the image, it returns an unevaluated ImageSet object.

$$f(x)|x \in self$$

See also:

[sympy.sets.fancysets.ImageSet](#) (page 1148)

Examples

```
>>> from sympy import S, Interval, Symbol, imageset, sin, Lambda
>>> from sympy.abc import x, y
```

```
>>> imageset(x, 2*x, Interval(0, 2))
Interval(0, 4)
```

```
>>> imageset(lambda x: 2*x, Interval(0, 2))
Interval(0, 4)
```

```
>>> imageset(Lambda(x, sin(x)), Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
```

```
>>> imageset(sin, Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
>>> imageset(lambda y: x + y, Interval(-2, 1))
ImageSet(Lambda(_x, _x + x), Interval(-2, 1))
```

Expressions applied to the set of Integers are simplified to show as few negatives as possible and linear expressions are converted to a canonical form. If this is not desirable then the unevaluated ImageSet should be used.

```
>>> imageset(x, -2*x + 5, S.Integers)
ImageSet(Lambda(x, 2*x + 1), Integers)
```

Elementary Sets

5.25.2 Interval

class sympy.sets.sets.Interval

Represents a real interval as a Set.

Usage: Returns an interval with end points “start” and “end”.

For left_open=True (default left_open is False) the interval will be open on the left. Similarly, for right_open=True the interval will be open on the right.

Notes

- Only real end points are supported
- Interval(a, b) with $a > b$ will return the empty set
- Use the evalf() method to turn an Interval into an mpmath ‘mpi’ interval instance

References

[R461] (page 1868)

Examples

```
>>> from sympy import Symbol, Interval
>>> Interval(0, 1)
Interval(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Lopen(0, 1)
Interval.Lopen(0, 1)
>>> Interval.open(0, 1)
Interval.open(0, 1)
```

```
>>> a = Symbol('a', real=True)
>>> Interval(0, a)
Interval(0, a)
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

classmethod Lopen(a, b)

Return an interval not including the left boundary.

classmethod Ropen(a, b)

Return an interval not including the right boundary.

as_relational(x)

Rewrite an interval in terms of inequalities and logic operators.

end

The right end point of 'self'.

This property takes the same value as the 'sup' property.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).end
1
```

is_left_unbounded

Return True if the left endpoint is negative infinity.

is_right_unbounded

Return True if the right endpoint is positive infinity.

left

The left end point of 'self'.

This property takes the same value as the 'inf' property.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```

left_open

True if 'self' is left-open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, left_open=True).left_open
True
>>> Interval(0, 1, left_open=False).left_open
False
```

classmethod open(a, b)

Return an interval including neither boundary.

right

The right end point of 'self'.

This property takes the same value as the 'sup' property.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).end
1
```

right_open

True if 'self' is right-open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, right_open=True).right_open
True
>>> Interval(0, 1, right_open=False).right_open
False
```

start

The left end point of 'self'.

This property takes the same value as the 'inf' property.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```

5.25.3 FiniteSet

class sympy.sets.sets.FiniteSet

Represents a finite set of discrete numbers

References

[R462] (page 1868)

Examples

```
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
>>> 3 in FiniteSet(1, 2, 3, 4)
True
```

```
>>> members = [1, 2, 3, 4]
>>> f = FiniteSet(*members)
>>> f
{1, 2, 3, 4}
>>> f - FiniteSet(2)
{1, 3, 4}
>>> f + FiniteSet(2, 5)
{1, 2, 3, 4, 5}
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

as_relational(symbol)

Rewrite a FiniteSet in terms of equalities and logic operators.

Compound Sets

5.25.4 Union

class sympy.sets.sets.Union

Represents a union of sets as a [Set](#) (page 1132).

See also:[Intersection](#) (page 1142)**References**[\[R463\]](#) (page 1868)**Examples**

```
>>> from sympy import Union, Interval
>>> Union(Interval(1, 2), Interval(3, 4))
Union(Interval(1, 2), Interval(3, 4))
```

The Union constructor will always try to merge overlapping intervals, if possible. For example:

```
>>> Union(Interval(1, 2), Interval(2, 3))
Interval(1, 3)
```

Attributes

<code>is_Complement</code>	
<code>is_EmptySet</code>	
<code>is_Intersection</code>	
<code>is_UniversalSet</code>	

as_relational(symbol)

Rewrite a Union in terms of equalities and logic operators.

5.25.5 Intersection

class `sympy.sets.sets.Intersection`

Represents an intersection of sets as a [Set](#) (page 1132).

See also:[Union](#) (page 1141)**References**[\[R464\]](#) (page 1868)**Examples**

```
>>> from sympy import Intersection, Interval
>>> Intersection(Interval(1, 3), Interval(2, 4))
Interval(2, 3)
```

We often use the `.intersect` method

```
>>> Interval(1,3).intersect(Interval(2,4))
Interval(2, 3)
```

Attributes

<code>is_Complement</code>	
<code>is_EmptySet</code>	
<code>is_UniversalSet</code>	

as_relational(symbol)

Rewrite an Intersection in terms of equalities and logic operators

5.25.6 ProductSet

class `sympy.sets.sets.ProductSet`

Represents a Cartesian Product of Sets.

Returns a Cartesian product given several sets as either an iterable or individual arguments.

Can use '*' operator on any sets for convenient shorthand.

Notes

- Passes most operations down to the argument sets
- Flattens Products of ProductSets

References

[R465] (page 1868)

Examples

```
>>> from sympy import Interval, FiniteSet, ProductSet
>>> I = Interval(0, 5); S = FiniteSet(1, 2, 3)
>>> ProductSet(I, S)
Interval(0, 5) x {1, 2, 3}
```

```
>>> (2, 2) in ProductSet(I, S)
True
```

```
>>> Interval(0, 1) * Interval(0, 1) # The unit square
Interval(0, 1) x Interval(0, 1)
```

```
>>> coin = FiniteSet('H', 'T')
>>> set(coin**2)
{(H, H), (H, T), (T, H), (T, T)}
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

is_iterable

A property method which tests whether a set is iterable or not. Returns True if set is iterable, otherwise returns False.

Examples

```
>>> from sympy import FiniteSet, Interval, ProductSet
>>> I = Interval(0, 1)
>>> A = FiniteSet(1, 2, 3, 4, 5)
>>> I.is_iterable
False
>>> A.is_iterable
True
```

5.25.7 Complement

class sympy.sets.sets.Complement

Represents the set difference or relative complement of a set with another set.

$$A - B = \{x \in A | x \notin B\}$$

See also:

[Intersection](#) (page 1142), [Union](#) (page 1141)

References

[R466] (page 1868)

Examples

```
>>> from sympy import Complement, FiniteSet
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
{0, 2}
```

Attributes

<code>is_EmptySet</code>	
<code>is_Intersection</code>	
<code>is_UniversalSet</code>	

static reduce(A, B)
Simplify a [Complement](#) (page 1144).

Singleton Sets

5.25.8 EmptySet

class `sympy.sets.sets.EmptySet`

Represents the empty set. The empty set is available as a singleton as `S.EmptySet`.

See also:

[UniversalSet](#) (page 1145)

References

[R467] (page 1869)

Examples

```
>>> from sympy import S, Interval
>>> S.EmptySet
EmptySet()
```

```
>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet()
```

Attributes

<code>is_Complement</code>	
<code>is_Intersection</code>	
<code>is_UniversalSet</code>	

5.25.9 UniversalSet

class `sympy.sets.sets.UniversalSet`

Represents the set of all things. The universal set is available as a singleton as `S.UniversalSet`

See also:

[EmptySet](#) (page 1145)

References

[R468] (page 1869)

Examples

```
>>> from sympy import S, Interval
>>> S.UniversalSet
UniversalSet()
```

```
>>> Interval(1, 2).intersect(S.UniversalSet)
Interval(1, 2)
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	

Special Sets

5.25.10 Naturals

class sympy.sets.fancysets.Naturals

Represents the natural numbers (or counting numbers) which are all positive integers starting from 1. This set is also available as the Singleton, S.Naturals.

See also:

Naturals0 (page 1147) non-negative integers (i.e. includes 0, too)

Integers (page 1147) also includes negative integers

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```


Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

5.25.11 Naturals0

class `sympy.sets.fancysets.Naturals0`

Represents the whole numbers which are all the non-negative integers, inclusive of zero.

See also:

Naturals (page 1146) positive integers; does not include 0

Integers (page 1147) also includes the negative integers

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

5.25.12 Integers

class `sympy.sets.fancysets.Integers`

Represents all integers: positive, negative and zero. This set is also available as the Singleton, `S.Integers`.

See also:

Naturals0 (page 1147) non-negative integers

Integers (page 1147) positive and negative integers and zero

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
>>> next(iterable)
-1
```

(continues on next page)

(continued from previous page)

```
>>> next(iterable)
2
```

```
>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

5.25.13 ImageSet

class sympy.sets.fancysets.ImageSet

Image of a set under a mathematical function. The transformation must be given as a Lambda function which has as many arguments as the elements of the set upon which it operates, e.g. 1 argument when acting on the set of integers or 2 arguments when acting on a complex region.

This function is not normally called directly, but is called from *imageset*.

See also:

[sympy.sets.sets.imageset](#) (page 1137)

Examples

```
>>> from sympy import Symbol, S, pi, Dummy, Lambda
>>> from sympy.sets.sets import FiniteSet, Interval
>>> from sympy.sets.fancysets import ImageSet
```

```
>>> x = Symbol('x')
>>> N = S.Naturals
>>> squares = ImageSet(Lambda(x, x**2), N) # {x**2 for x in N}
>>> 4 in squares
True
>>> 5 in squares
False
```

```
>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersect(squares)
{1, 4, 9}
```

```
>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
4
```

(continues on next page)

(continued from previous page)

```
9
16
```

If you want to get value for $x = 2, 1/2$ etc. (Please check whether the x value is in *base_{set}* or not before passing it as args)

```
>>> squares.lamda(2)
4
>>> squares.lamda(S(1)/2)
1/4
```

```
>>> n = Dummy('n')
>>> solutions = ImageSet(Lambda(n, n*pi), S.Integers) # solutions of sin(x) = 0
>>> dom = Interval(-1, 1)
>>> dom.intersect(solutions)
{0}
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

5.25.14 Range

class sympy.sets.fancysets.Range

Represents a range of integers. Can be called as `Range(stop)`, `Range(start, stop)`, or `Range(start, stop, step)`; when stop is not given it defaults to 1.

Range(stop) is the same as *Range(0, stop, 1)* and the stop value (juse as for Python ranges) is not included in the Range values.

```
>>> from sympy import Range
>>> list(Range(3))
[0, 1, 2]
```

The step can also be negative:

```
>>> list(Range(10, 0, -2))
[10, 8, 6, 4, 2]
```

The stop value is made canonical so equivalent ranges always have the same args:

```
>>> Range(0, 10, 3)
Range(0, 12, 3)
```

Infinite ranges are allowed. If the starting point is infinite, then the final value is `stop - step`. To iterate such a range, it needs to be reversed:

```
>>> from sympy import oo
>>> r = Range(-oo, 1)
>>> r[-1]
0
>>> next(iter(r))
Traceback (most recent call last):
...
ValueError: Cannot iterate over Range with infinite start
>>> next(iter(r.reversed))
0
```

Although `Range` is a set (and supports the normal set operations) it maintains the order of the elements and can be used in contexts where *range* would be used.

```
>>> from sympy import Interval
>>> Range(0, 10, 2).intersect(Interval(3, 7))
Range(4, 8, 2)
>>> list(_)
[4, 6]
```

Although slicing of a `Range` will always return a `Range` - possibly empty - an empty set will be returned from any intersection that is empty:

```
>>> Range(3)[:0]
Range(0, 0, 1)
>>> Range(3).intersect(Interval(4, oo))
EmptySet()
>>> Range(3).intersect(Range(4, oo))
EmptySet()
```

Attributes

<code>is_Complement</code>	
<code>is_EmptySet</code>	
<code>is_Intersection</code>	
<code>is_UniversalSet</code>	

reversed

Return an equivalent `Range` in the opposite order.

Examples

```
>>> from sympy import Range
>>> Range(10).reversed
Range(9, -1, -1)
```

5.25.15 ComplexRegion

`class sympy.sets.fancysets.ComplexRegion`

Represents the Set of all Complex Numbers. It can represent a region of Complex Plane in both the standard forms Polar and Rectangular coordinates.

- Polar Form Input is in the form of the ProductSet or Union of ProductSets of the intervals of r and θ , & use the flag `polar=True`.

$Z = \{z \in \mathbb{C} \mid z = r[\cos(\theta) + I\sin(\theta)], r \in [r], \theta \in [\theta]\}$

- Rectangular Form Input is in the form of the ProductSet or Union of ProductSets of interval of x and y the of the Complex numbers in a Plane. Default input type is in rectangular form.

$Z = \{z \in \mathbb{C} \mid z = x + I*y, x \in [\operatorname{Re}(z)], y \in [\operatorname{Im}(z)]\}$

See also:

Reals

Examples

```
>>> from sympy.sets.fancysets import ComplexRegion
>>> from sympy.sets import Interval
>>> from sympy import S, I, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 6)
>>> c = Interval(1, 8)
>>> c1 = ComplexRegion(a*b) # Rectangular Form
>>> c1
ComplexRegion(Interval(2, 3) x Interval(4, 6), False)
```

- `c1` represents the rectangular region in complex plane surrounded by the coordinates (2, 4), (3, 4), (3, 6) and (2, 6), of the four vertices.

```
>>> c2 = ComplexRegion(Union(a*b, b*c))
>>> c2
ComplexRegion(Union(Interval(2, 3) x Interval(4, 6), Interval(4, 6) x Interval(1, 8)), False)
```

- `c2` represents the Union of two rectangular regions in complex plane. One of them surrounded by the coordinates of `c1` and other surrounded by the coordinates (4, 1), (6, 1), (6, 8) and (4, 8).

```
>>> 2.5 + 4.5*I in c1
True
>>> 2.5 + 6.5*I in c1
False
```

```
>>> r = Interval(0, 1)
>>> theta = Interval(0, 2*S.Pi)
>>> c2 = ComplexRegion(r*theta, polar=True) # Polar Form
>>> c2 # unit Disk
ComplexRegion(Interval(0, 1) x Interval.Ropen(0, 2*pi), True)
```

- `c2` represents the region in complex plane inside the Unit Disk centered at the origin.

```
>>> 0.5 + 0.5*I in c2
True
```

(continues on next page)

(continued from previous page)

```
>>> 1 + 2*I in c2
False
```

```
>>> unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, 2*S.Pi), polar=True)
>>> upper_half_unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, S.Pi),
↳polar=True)
>>> intersection = unit_disk.intersect(upper_half_unit_disk)
>>> intersection
ComplexRegion(Interval(0, 1) x Interval(0, pi), True)
>>> intersection == upper_half_unit_disk
True
```

Attributes

is_Complement	
is_EmptySet	
is_Intersection	
is_UniversalSet	

a_interval

Return the union of intervals of x when, self is in rectangular form, or the union of intervals of r when self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.a_interval
Interval(2, 3)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.a_interval
Union(Interval(2, 3), Interval(4, 5))
```

b_interval

Return the union of intervals of y when, self is in rectangular form, or the union of intervals of θ when self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.b_interval
```

(continues on next page)

(continued from previous page)

```
Interval(4, 5)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.b_interval
Interval(1, 7)
```

classmethod from_real(sets)

Converts given subset of real numbers to a complex region.

Examples

```
>>> from sympy import Interval, ComplexRegion
>>> unit = Interval(0,1)
>>> ComplexRegion.from_real(unit)
ComplexRegion(Interval(0, 1) x {0}, False)
```

polar

Returns True if self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union, S
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> theta = Interval(0, 2*S.Pi)
>>> C1 = ComplexRegion(a*b)
>>> C1.polar
False
>>> C2 = ComplexRegion(a*theta, polar=True)
>>> C2.polar
True
```

psets

Return a tuple of sets (ProductSets) input of the self.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.psets
(Interval(2, 3) x Interval(4, 5),)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.psets
(Interval(2, 3) x Interval(4, 5), Interval(4, 5) x Interval(1, 7))
```

sets

Return raw input sets to the self.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.sets
Interval(2, 3) x Interval(4, 5)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.sets
Union(Interval(2, 3) x Interval(4, 5), Interval(4, 5) x Interval(1, 7))
```

`sympy.sets.fancysets.normalize_theta_set(theta)`

Normalize a Real Set *theta* in the Interval $[0, 2\pi]$. It returns a normalized value of *theta* in the Set. For Interval, a maximum of one cycle $[0, 2\pi]$, is returned i.e. for *theta* equal to $[0, 10\pi]$, returned normalized value would be $[0, 2\pi]$. As of now intervals with end points as non-multiples of π is not supported.

Raises NotImplementedError

The algorithms for Normalizing *theta* Set are not yet implemented.

ValueError

The input is not valid, i.e. the input is not a real set.

RuntimeError

It is a bug, please report to the github issue tracker.

Examples

```
>>> from sympy.sets.fancysets import normalize_theta_set
>>> from sympy import Interval, FiniteSet, pi
>>> normalize_theta_set(Interval(9*pi/2, 5*pi))
Interval(pi/2, pi)
>>> normalize_theta_set(Interval(-3*pi/2, pi/2))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-pi/2, pi/2))
Union(Interval(0, pi/2), Interval.Ropen(3*pi/2, 2*pi))
>>> normalize_theta_set(Interval(-4*pi, 3*pi))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-3*pi/2, -pi/2))
Interval(pi/2, 3*pi/2)
>>> normalize_theta_set(FiniteSet(0, pi, 3*pi))
{0, pi}
```

5.26 Simplify

5.26.1 simplify

`sympy.simplify.simplify.simplify(expr, ratio=1.7, measure=<function count_ops>, rational=False, inverse=False)`

Simplifies the given expression.

Simplification is not a well defined term and the exact strategies this function tries can change in the future versions of SymPy. If your algorithm relies on “simplification” (whatever it is), try to determine what you need exactly - is it `powsimp()`?, `radsimp()`?, `together()`?, `logcombine()`?, or something else? And use this particular function directly, because those are well defined and thus your algorithm will be robust.

Nonetheless, especially for interactive use, or when you don’t know anything about the structure of the expression, `simplify()` tries to apply intelligent heuristics to make the input expression “simpler”. For example:

```
>>> from sympy import simplify, cos, sin
>>> from sympy.abc import x, y
>>> a = (x + x**2)/(x*sin(y)**2 + x*cos(y)**2)
>>> a
(x**2 + x)/(x*sin(y)**2 + x*cos(y)**2)
>>> simplify(a)
x + 1
```

Note that we could have obtained the same result by using specific simplification functions:

```
>>> from sympy import trigsimp, cancel
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
x + 1
```

In some cases, applying `simplify()` (page 1154) may actually result in some more complicated expression. The default `ratio=1.7` prevents more extreme cases: if $(\text{result length})/(\text{input length}) > \text{ratio}$, then input is returned unmodified. The `measure` parameter lets you specify the function used to determine how complex an expression is. The function should take a single argument as an expression and return a number such that if expression `a` is more complex than expression `b`, then `measure(a) > measure(b)`. The default measure function is `count_ops()`, which returns the total number of operations in the expression.

For example, if `ratio=1`, `simplify` output can’t be longer than input.

```
>>> from sympy import sqrt, simplify, count_ops, oo
>>> root = 1/(sqrt(2)+3)
```

Since `simplify(root)` would result in a slightly longer expression, `root` is returned unchanged instead:

```
>>> simplify(root, ratio=1) == root
True
```

If `ratio=oo`, `simplify` will be applied anyway:

```
>>> count_ops(simplify(root, ratio=oo)) > count_ops(root)
True
```

Note that the shortest expression is not necessary the simplest, so setting `ratio` to 1 may not be a good idea. Heuristically, the default value `ratio=1.7` seems like a reasonable choice.

You can easily define your own measure function based on what you feel should represent the “size” or “complexity” of the input expression. Note that some choices, such as `lambda expr: len(str(expr))` may appear to be good metrics, but have other problems

(in this case, the measure function may slow down simplify too much for very large expressions). If you don't know what a good metric would be, the default, `count_ops`, is a good one.

For example:

```
>>> from sympy import symbols, log
>>> a, b = symbols('a b', positive=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
>>> h
log(a*b**(-log(a) + 1))
>>> count_ops(g)
8
>>> count_ops(h)
5
```

So you can see that `h` is simpler than `g` using the `count_ops` metric. However, we may not like how `simplify` (in this case, using `logcombine`) has created the `b**(log(1/a) + 1)` term. A simple way to reduce this would be to give more weight to powers as operations in `count_ops`. We can do this by using the `visual=True` option:

```
>>> print(count_ops(g, visual=True))
2*ADD + DIV + 4*LOG + MUL
>>> print(count_ops(h, visual=True))
2*LOG + MUL + POW + SUB
```

```
>>> from sympy import Symbol, S
>>> def my_measure(expr):
...     POW = Symbol('POW')
...     # Discourage powers by giving POW a weight of 10
...     count = count_ops(expr, visual=True).subs(POW, 10)
...     # Every other operation gets a weight of 1 (the default)
...     count = count.replace(Symbol, type(S.One))
...     return count
>>> my_measure(g)
8
>>> my_measure(h)
14
>>> 15./8 > 1.7 # 1.7 is the default ratio
True
>>> simplify(g, measure=my_measure)
-log(a)*log(b) + log(a) + log(b)
```

Note that because `simplify()` internally tries many different simplification strategies and then compares them using the measure function, we get a completely different result that is still different from the input expression by doing this.

If `rational=True`, Floats will be recast as Rationals before simplification. If `rational=None`, Floats will be recast as Rationals but the result will be recast as Floats. If `rational=False` (default) then nothing will be done to the Floats.

If `inverse=True`, it will be assumed that a composition of inverse functions, such as `sin` and `asin`, can be cancelled in any order. For example, `asin(sin(x))` will yield `x` without checking whether `x` belongs to the set where this relation is true. The default is `False`.

5.26.2 separatevars

`sympy.simplify.simplify.separatevars(expr, symbols=[], dict=False, force=False)`

Separates variables in an expression, if possible. By default, it separates with respect to all symbols in an expression and collects constant coefficients that are independent of symbols.

If `dict=True` then the separated terms will be returned in a dictionary keyed to their corresponding symbols. By default, all symbols in the expression will appear as keys; if symbols are provided, then all those symbols will be used as keys, and any terms in the expression containing other symbols or non-symbols will be returned keyed to the string 'coeff'. (Passing `None` for symbols will return the expression in a dictionary keyed to 'coeff'.)

If `force=True`, then bases of powers will be separated regardless of assumptions on the symbols involved.

Notes

The order of the factors is determined by `Mul`, so that the separated expressions may not necessarily be grouped together.

Although factoring is necessary to separate variables in some expressions, it is not necessary in all cases, so one should not count on the returned factors being factored.

Examples

```
>>> from sympy.abc import x, y, z, alpha
>>> from sympy import separatevars, sin
>>> separatevars((x*y)**y)
(x*y)**y
>>> separatevars((x*y)**y, force=True)
x**y*y**y
```

```
>>> e = 2*x**2*z*sin(y)+2*z*x**2
>>> separatevars(e)
2*x**2*z*(sin(y) + 1)
>>> separatevars(e, symbols=(x, y), dict=True)
{'coeff': 2*z, x: x**2, y: sin(y) + 1}
>>> separatevars(e, [x, y, alpha], dict=True)
{'coeff': 2*z, alpha: 1, x: x**2, y: sin(y) + 1}
```

If the expression is not really separable, or is only partially separable, `separatevars` will do the best it can to separate it by using factoring.

```
>>> separatevars(x + x*y - 3*x**2)
-x*(3*x - y - 1)
```

If the expression is not separable then `expr` is returned unchanged or (if `dict=True`) then `None` is returned.

```
>>> eq = 2*x + y*sin(x)
>>> separatevars(eq) == eq
True
```

(continues on next page)

(continued from previous page)

```
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) == None
True
```

5.26.3 nthroot

`sympy.simplify.simplify.nthroot(expr, n, max_len=4, prec=15)`

compute a real nth-root of a sum of surds

Parameters `expr` : sum of surds

`n` : integer

`max_len` : maximum number of surds passed as constants to `nsimplify`

Examples

```
>>> from sympy.simplify.simplify import nthroot
>>> from sympy import Rational, sqrt
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

Algorithm

First `nsimplify` is used to get a candidate root; if it is not a root the minimal polynomial is computed; the answer is one of its roots.

5.26.4 besselsimp

`sympy.simplify.simplify.besselsimp(expr)`

Simplify `bessel`-type functions.

This routine tries to simplify `bessel`-type functions. Currently it only works on the Bessel J and I functions, however. It works by looking at all such functions in turn, and eliminating factors of “I” and “-1” (actually their polar equivalents) in front of the argument. Then, functions of half-integer order are rewritten using trigonometric functions and functions of integer order (> 1) are rewritten using functions of low order. Finally, if the expression was changed, compute factorization of the result with `factor()`.

```
>>> from sympy import besselj, besseli, besselsimp, polar_lift, I, S
>>> from sympy.abc import z, nu
>>> besselsimp(besselj(nu, z*polar_lift(-1)))
exp(I*pi*nu)*besselj(nu, z)
>>> besselsimp(besseli(nu, z*polar_lift(-I)))
exp(-I*pi*nu/2)*besselj(nu, z)
>>> besselsimp(besseli(S(-1)/2, z))
sqrt(2)*cosh(z)/(sqrt(pi)*sqrt(z))
>>> besselsimp(z*besseli(0, z) + z*(besseli(2, z))/2 + besseli(1, z))
3*z*besseli(0, z)/2
```

5.26.5 hypersimp

`sympy.simplify.simplify.hypersimp(f, k)`

Given combinatorial term $f(k)$ simplify its consecutive term ratio i.e. $f(k+1)/f(k)$. The input term can be composed of functions and integer sequences which have equivalent representation in terms of gamma special function.

The algorithm performs three basic steps:

1. Rewrite all functions in terms of gamma, if possible.
2. Rewrite all occurrences of gamma in terms of products of gamma and rising factorial with integer, absolute constant exponent.
3. Perform simplification of nested fractions, powers and if the resulting expression is a quotient of polynomials, reduce their total degree.

If $f(k)$ is hypergeometric then as result we arrive with a quotient of polynomials of minimal degree. Otherwise None is returned.

For more information on the implemented algorithm refer to:

1. W. Koepf, Algorithms for m-fold Hypergeometric Summation, Journal of Symbolic Computation (1995) 20, 399-417

5.26.6 hypersimilar

`sympy.simplify.simplify.hypersimilar(f, g, k)`

Returns True if 'f' and 'g' are hyper-similar.

Similarity in hypergeometric sense means that a quotient of $f(k)$ and $g(k)$ is a rational function in k . This procedure is useful in solving recurrence relations.

For more information see `hypersimp()`.

5.26.7 nsimplify

`sympy.simplify.simplify.nsimplify(expr, constants=(), tolerance=None, full=False, rational=None, rational_conversion='base10')`

Find a simple representation for a number or, if there are free symbols or if `rational=True`, then replace Floats with their Rational equivalents. If no change is made and `rational` is not False then Floats will at least be converted to Rationals.

For numerical expressions, a simple formula that numerically matches the given numerical expression is sought (and the input should be possible to evalf to a precision of at least 30 digits).

Optionally, a list of (rationally independent) constants to include in the formula may be given.

A lower tolerance may be set to find less exact matches. If no tolerance is given then the least precise value will set the tolerance (e.g. Floats default to 15 digits of precision, so would be `tolerance=10**-15`).

With `full=True`, a more extensive search is performed (this is useful to find simpler numbers when the tolerance is set low).

When converting to rational, if `rational_conversion='base10'` (the default), then convert floats to rationals using their base-10 (string) representation. When `rational_conversion='exact'` it uses the exact, base-2 representation.

See also:

`sympy.core.function.nfloat` (page 196)

Examples

```
>>> from sympy import nsimplify, sqrt, GoldenRatio, exp, I, pi
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1/2 - I*sqrt(sqrt(5)/10 + 1/4)
>>> nsimplify(I*I, [pi])
exp(-pi/2)
>>> nsimplify(pi, tolerance=0.01)
22/7
```

```
>>> nsimplify(0.333333333333333, rational=True, rational_conversion='exact')
6004799503160655/18014398509481984
>>> nsimplify(0.333333333333333, rational=True)
1/3
```

5.26.8 posify

`sympy.simplify.simplify.posify(eq)`

Return `eq` (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.

Any symbol that has `positive=None` will be replaced with a positive dummy symbol having the same name. This replacement will allow more symbolic processing of expressions, especially those involving powers and logarithms.

A dictionary that can be sent to `subs` to restore `eq` to its original symbols is also returned.

```
>>> from sympy import posify, Symbol, log, solve
>>> from sympy.abc import x
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(_x + n + p, {_x: x})
```

```
>>> eq = 1/x
>>> log(eq).expand()
log(1/x)
>>> log(posify(eq)[0]).expand()
-log(_x)
>>> p, rep = posify(eq)
>>> log(p).expand().subs(rep)
-log(x)
```

It is possible to apply the same transformations to an iterable of expressions:

```
>>> eq = x**2 - 4
>>> solve(eq, x)
[-2, 2]
>>> eq_x, reps = posify([eq, x]); eq_x
[_x**2 - 4, _x]
>>> solve(*eq_x)
[2]
```

5.26.9 logcombine

`sympy.simplify.simplify.logcombine(expr, force=False)`

Takes logarithms and combines them using the following rules:

- $\log(x) + \log(y) == \log(x*y)$ if both are not negative
- $a*\log(x) == \log(x**a)$ if x is positive and a is real

If `force` is `True` then the assumptions above will be assumed to hold if there is no assumption already in place on a quantity. For example, if a is imaginary or the argument negative, `force` will not perform a combination but if a is a symbol with no assumptions the change will take place.

See also:

posify (page 1160) replace all symbols with symbols having positive assumptions

sympy.core.function.expand_log (page 193) expand the logarithms of products and powers; the opposite of `logcombine`

Examples

```
>>> from sympy import Symbol, symbols, log, logcombine, I
>>> from sympy.abc import a, x, y, z
>>> logcombine(a*log(x) + log(y) - log(z))
a*log(x) + log(y) - log(z)
>>> logcombine(a*log(x) + log(y) - log(z), force=True)
log(x**a*y/z)
>>> x,y,z = symbols('x,y,z', positive=True)
>>> a = Symbol('a', real=True)
>>> logcombine(a*log(x) + log(y) - log(z))
log(x**a*y/z)
```

The transformation is limited to factors and/or terms that contain logs, so the result depends on the initial state of expansion:

```
>>> eq = (2 + 3*I)*log(x)
>>> logcombine(eq, force=True) == eq
True
>>> logcombine(eq.expand(), force=True)
log(x**2) + I*log(x**3)
```

5.26.10 Radsimp

radsimp

`sympy.simplify.radsimp.radsimp(expr, symbolic=True, max_terms=4)`

Rationalize the denominator by removing square roots.

Note: the expression returned from `radsimp` must be used with caution since if the denominator contains symbols, it will be possible to make substitutions that violate the assumptions of the simplification process: that for a denominator matching $a + b\sqrt{c}$, $a \neq \pm b\sqrt{c}$. (If there are no symbols, this assumption is made valid by collecting terms of $\sqrt{c}$ so the match variable a does not contain $\sqrt{c}$.) If you do not want the simplification to occur for symbolic denominators, set `symbolic` to `False`.

If there are more than `max_terms` radical terms then the expression is returned unchanged.

Examples

```
>>> from sympy import radsimp, sqrt, Symbol, denom, pprint, I
>>> from sympy import factor_terms, fraction, signsimp
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b, c
```

```
>>> radsimp(1/(2 + sqrt(2)))
(-sqrt(2) + 2)/2
>>> x,y = map(Symbol, 'xy')
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)
```

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5)); pprint(ans)
```

$$\frac{\sqrt{5}a + \sqrt{5}b - \sqrt{2}x - \sqrt{2}y}{5a^2 + 10ab + 5b^2 - 2x^2 - 4xy - 2y^2}$$

```
>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))
```

$$\frac{\sqrt{5}(a + b) - \sqrt{2}(x + y)}{5a^2 + 10ab + 5b^2 - 2x^2 - 4xy - 2y^2}$$

If radicals in the denominator cannot be removed or there is no denominator, the original expression will be returned.


```
>>> radsimp(sqrt(2)*x + sqrt(2))
sqrt(2)*x + sqrt(2)
```

Results with symbols will not always be valid for all substitutions:

```
>>> eq = 1/(a + b*sqrt(c))
>>> eq.subs(a, b*sqrt(c))
1/(2*b*sqrt(c))
>>> radsimp(eq).subs(a, b*sqrt(c))
nan
```

If `symbolic=False`, symbolic denominators will not be transformed (but numeric denominators will still be processed):

```
>>> radsimp(eq, symbolic=False)
1/(a + b*sqrt(c))
```

rad_rationalize

`sympy.simplify.radsimp.rad_rationalize(num, den)`

Rationalize num/den by removing square roots in the denominator; num and den are sum of terms whose squares are rationals

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import rad_rationalize
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

collect

`sympy.simplify.radsimp.collect(expr, syms, func=None, evaluate=None, exact=False, distribute_order_term=True)`

Collect additive terms of an expression.

This function collects additive terms of an expression with respect to a list of expression up to powers with rational exponents. By the term symbol here are meant arbitrary expressions, which can contain powers, products, sums etc. In other words symbol is a pattern which will be searched for in the expression's terms.

The input expression is not expanded by `collect()` (page 1163), so user is expected to provide an expression in an appropriate form. This makes `collect()` (page 1163) more predictable as there is no magic happening behind the scenes. However, it is important to note, that powers of products are converted to products of powers using the `expand_power_base()` function.

There are two possible types of output. First, if `evaluate` flag is set, this function will return an expression with collected terms or else it will return a dictionary with expressions up to rational powers as keys and collected coefficients as values.

See also:

`collect_const` (page 1166), `collect_sqrt` (page 1166), `rcollect` (page 1165)

Examples

```
>>> from sympy import S, collect, expand, factor, Wild
>>> from sympy.abc import a, b, c, x, y, z
```

This function can collect symbolic coefficients in polynomials or rational expressions. It will manage to find all integer or rational powers of collection variable:

```
>>> collect(a*x**2 + b*x**2 + a*x - b*x + c, x)
c + x**2*(a + b) + x*(a - b)
```

The same result can be achieved in dictionary form:

```
>>> d = collect(a*x**2 + b*x**2 + a*x - b*x + c, x, evaluate=False)
>>> d[x**2]
a + b
>>> d[x]
a - b
>>> d[S.One]
c
```

You can also work with multivariate polynomials. However, remember that this function is greedy so it will care only about a single symbol at time, in specification order:

```
>>> collect(x**2 + y*x**2 + x*y + y + a*y, [x, y])
x**2*(y + 1) + x*y + y*(a + 1)
```

Also more complicated expressions can be used as patterns:

```
>>> from sympy import sin, log
>>> collect(a*sin(2*x) + b*sin(2*x), sin(2*x))
(a + b)*sin(2*x)

>>> collect(a*x*log(x) + b*(x*log(x)), x*log(x))
x*(a + b)*log(x)
```

You can use wildcards in the pattern:

```
>>> w = Wild('w1')
>>> collect(a*x**y - b*x**y, w**y)
x**y*(a - b)
```

It is also possible to work with symbolic powers, although it has more complicated behavior, because in this case power's base and symbolic part of the exponent are treated as a single symbol:

```
>>> collect(a*x**c + b*x**c, x)
a*x**c + b*x**c
>>> collect(a*x**c + b*x**c, x**c)
x**c*(a + b)
```

However if you incorporate rationals to the exponents, then you will get well known behavior:

```
>>> collect(a*x**(2*c) + b*x**(2*c), x**c)
x**(2*c)*(a + b)
```

Note also that all previously stated facts about `collect()` (page 1163) function apply to the exponential function, so you can get:

```
>>> from sympy import exp
>>> collect(a*exp(2*x) + b*exp(2*x), exp(x))
(a + b)*exp(2*x)
```

If you are interested only in collecting specific powers of some symbols then set `exact` flag in arguments:

```
>>> collect(a*x**7 + b*x**7, x, exact=True)
a*x**7 + b*x**7
>>> collect(a*x**7 + b*x**7, x**7, exact=True)
x**7*(a + b)
```

You can also apply this function to differential equations, where derivatives of arbitrary order can be collected. Note that if you collect with respect to a function or a derivative of a function, all derivatives of that function will also be collected. Use `exact=True` to prevent this from happening:

```
>>> from sympy import Derivative as D, collect, Function
>>> f = Function('f')(x)

>>> collect(a*D(f,x) + b*D(f,x), D(f,x))
(a + b)*Derivative(f(x), x)

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), f)
(a + b)*Derivative(f(x), (x, 2))

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
a*Derivative(f(x), (x, 2)) + b*Derivative(f(x), (x, 2))

>>> collect(a*D(f,x) + b*D(f,x) + a*f + b*f, f)
(a + b)*f(x) + (a + b)*Derivative(f(x), x)
```

Or you can even match both derivative order and exponent at the same time:

```
>>> collect(a*D(D(f,x),x)**2 + b*D(D(f,x),x)**2, D(f,x))
(a + b)*Derivative(f(x), (x, 2))**2
```

Finally, you can apply a function to each of the collected coefficients. For example you can factorize symbolic coefficients of polynomial:

```
>>> f = expand((x + a + 1)**3)

>>> collect(f, x, factor)
x**3 + 3*x**2*(a + 1) + 3*x*(a + 1)**2 + (a + 1)**3
```

Note: Arguments are expected to be in expanded form, so you might have to call `expand()` prior to calling this function.

`sympy.simplify.radsimp.rcollect(expr, *vars)`
 Recursively collect sums in an expression.

See also:

`collect` (page 1163), `collect_const` (page 1166), `collect_sqrt` (page 1166)

Examples

```
>>> from sympy.simplify import rcollect
>>> from sympy.abc import x, y
```

```
>>> expr = (x**2*y + x*y + x + y)/(x + y)
```

```
>>> rcollect(expr, y)
(x + y*(x**2 + x + 1))/(x + y)
```

collect_sqrt

`sympy.simplify.radsimp.collect_sqrt(expr, evaluate=None)`

Return `expr` with terms having common square roots collected together. If `evaluate` is `False` a count indicating the number of `sqrt`-containing terms will be returned and, if non-zero, the terms of the Add will be returned, else the expression itself will be returned as a single term. If `evaluate` is `True`, the expression with any collected terms will be returned.

Note: since $I = \text{sqrt}(-1)$, it is collected, too.

See also:

[collect](#) (page 1163), [collect_const](#) (page 1166), [rcollect](#) (page 1165)

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b
```

```
>>> r2, r3, r5 = [sqrt(i) for i in [2, 3, 5]]
>>> collect_sqrt(a*r2 + b*r2)
sqrt(2)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r3)
sqrt(2)*(a + b) + sqrt(3)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5)
sqrt(3)*a + sqrt(5)*b + sqrt(2)*(a + b)
```

If `evaluate` is `False` then the arguments will be sorted and returned as a list and a count of the number of `sqrt`-containing terms will be returned:

```
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5, evaluate=False)
((sqrt(3)*a, sqrt(5)*b, sqrt(2)*(a + b)), 3)
>>> collect_sqrt(a*sqrt(2) + b, evaluate=False)
((b, sqrt(2)*a), 1)
>>> collect_sqrt(a + b, evaluate=False)
((a + b,), 0)
```

collect_const

`sympy.simplify.radsimp.collect_const(expr, *vars, **kwargs)`

A non-greedy collection of terms with similar number coefficients in an Add `expr`. If

`vars` is given then only those constants will be targeted. Although any `Number` can also be targeted, if this is not desired set `Numbers=False` and no `Float` or `Rational` will be collected.

Parameters `expr` : sympy expression

This parameter defines the expression the expression from which terms with similar coefficients are to be collected. A non-Add expression is returned as it is.

vars : variable length collection of `Numbers`, optional

Specifies the constants to target for collection. Can be multiple in number.

kwargs : `Numbers` is the only possible argument to pass.

`Numbers` (default=True) specifies to target all instance of `sympy.core.numbers.Number` (page 138) class. If `Numbers=False`, then no `Float` or `Rational` will be collected.

Returns `expr` : `Expr`

Returns an expression with similar coefficient terms collected.

See also:

`collect` (page 1163), `collect_sqrt` (page 1166), `rcollect` (page 1165)

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import a, s, x, y, z
>>> from sympy.simplify.radsimp import collect_const
>>> collect_const(sqrt(3) + sqrt(3)*(1 + sqrt(2)))
sqrt(3)*(sqrt(2) + 2)
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(3) + sqrt(7))
(sqrt(3) + sqrt(7))*(s + 1)
>>> s = sqrt(2) + 2
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7))
(sqrt(2) + 3)*(sqrt(3) + sqrt(7))
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7), sqrt(3))
sqrt(7) + sqrt(3)*(sqrt(2) + 3) + sqrt(7)*(sqrt(2) + 2)
```

The collection is sign-sensitive, giving higher precedence to the unsigned values:

```
>>> collect_const(x - y - z)
x - (y + z)
>>> collect_const(-y - z)
-(y + z)
>>> collect_const(2*x - 2*y - 2*z, 2)
2*(x - y - z)
>>> collect_const(2*x - 2*y - 2*z, -2)
2*x - 2*(y + z)
```

fraction

`sympy.simplify.radsimp.fraction(expr, exact=False)`

Returns a pair with expression's numerator and denominator. If the given expression is

not a fraction then this function will return the tuple (expr, 1).

This function will not make any attempt to simplify nested fractions or to do any term rewriting at all.

If only one of the numerator/denominator pair is needed then use `numer(expr)` or `denom(expr)` functions respectively.

```
>>> from sympy import fraction, Rational, Symbol
>>> from sympy.abc import x, y
```

```
>>> fraction(x/y)
(x, y)
>>> fraction(x)
(x, 1)
```

```
>>> fraction(1/y**2)
(1, y**2)
```

```
>>> fraction(x*y/2)
(x*y, 2)
>>> fraction(Rational(1, 2))
(1, 2)
```

This function will also work fine with assumptions:

```
>>> k = Symbol('k', negative=True)
>>> fraction(x * y**k)
(x, y**(-k))
```

If we know nothing about sign of some exponent and 'exact' flag is unset, then structure this exponent's structure will be analyzed and pretty fraction will be returned:

```
>>> from sympy import exp, Mul
>>> fraction(2*x**(-y))
(2, x**y)
```

```
>>> fraction(exp(-x))
(1, exp(x))
```

```
>>> fraction(exp(-x), exact=True)
(exp(-x), 1)
```

The *exact* flag will also keep any unevaluated Muls from being evaluated:

```
>>> u = Mul(2, x + 1, evaluate=False)
>>> fraction(u)
(2*x + 2, 1)
>>> fraction(u, exact=True)
(2*(x + 1), 1)
```

5.26.11 Ratsimp

ratsimp

`sympy.simplify.ratsimp.ratsimp(expr)`

Put an expression over a common denominator, cancel and reduce.

Examples

```
>>> from sympy import ratsimp
>>> from sympy.abc import x, y
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)
```

5.26.12 Trigonometric simplification

trigsimp

`sympy.simplify.trigsimp.trigsimp(expr, **opts)`

reduces expression by using known trig identities

Notes

method: - Determine the method to use. Valid choices are 'matching' (default), 'groebner', 'combined', and 'fu'. If 'matching', simplify the expression recursively by targeting common patterns. If 'groebner', apply an experimental groebner basis algorithm. In this case further options are forwarded to `trigsimp_groebner`, please refer to its docstring. If 'combined', first run the groebner basis algorithm with small default parameters, then run the 'matching' algorithm. 'fu' runs the collection of trigonometric transformations described by Fu, et al. (see the *fu* docstring).

Examples

```
>>> from sympy import trigsimp, sin, cos, log
>>> from sympy.abc import x, y
>>> e = 2*sin(x)**2 + 2*cos(x)**2
>>> trigsimp(e)
2
```

Simplification occurs wherever trigonometric functions are located.

```
>>> trigsimp(log(e))
log(2)
```

Using `method = "groebner"` (or `"combined"`) might lead to greater simplification.

The old `trigsimp` routine can be accessed as with method 'old'.

```
>>> from sympy import coth, tanh
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

5.26.13 Power simplification

powsimp

`sympy.simplify.powsimp.powsimp(expr, deep=False, combine='all', force=False, measure=<function count_ops>)`
reduces expression by combining powers with similar bases and exponents.

Notes

If `deep` is `True` then `powsimp()` will also simplify arguments of functions. By default `deep` is set to `False`.

If `force` is `True` then bases will be combined without checking for assumptions, e.g. $\sqrt{x}*\sqrt{y} \rightarrow \sqrt{x*y}$ which is not true if x and y are both negative.

You can make `powsimp()` only combine bases or only combine exponents by changing `combine='base'` or `combine='exp'`. By default, `combine='all'`, which does both. `combine='base'` will only combine:

```
a   a           a           2x      x
x * y => (x*y)  as well as things like 2  => 4
```

and `combine='exp'` will only combine

```
a   b      (a + b)
x * x => x
```

`combine='exp'` will strictly only combine exponents in the way that used to be automatic. Also use `deep=True` if you need the old behavior.

When `combine='all'`, `'exp'` is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want `'base'` combined first, do something like `powsimp(powsimp(expr, combine='base'), combine='exp')`.

Examples

```
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```



```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> from sympy import sqrt, Mul
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _*a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

powdenest

`sympy.simplify.powsimp.powdenest(eq, force=False, polar=False)`

Collect exponents on powers as assumptions allow.

Given $(bb**be)**e$, this can be simplified as follows:

- if bb is positive, or
- e is an integer, or
- $|be| < 1$ then this simplifies to $bb**(be*e)$

Given a product of powers raised to a power, $(bb1**be1 * bb2**be2...)**e$, simplification can be done as follows:

- if e is positive, the gcd of all bei can be joined with e ;
- all non-negative bb can be separated from those that are negative and their gcd can be joined with e ; autosimplification already handles this separation.
- integer factors from powers that have integers in the denominator of the exponent can be removed from any term and the gcd of such integers can be joined with e

Setting `force` to `True` will make symbols that are not explicitly negative behave as though they are positive, resulting in more denesting.

Setting `polar` to `True` will do simplifications on the Riemann surface of the logarithm, also resulting in more denestings.

When there are sums of logs in `exp()` then a product of powers may be obtained e.g. `exp(3*(log(a) + 2*log(b))) -> a**3*b**6`.

Examples

```
>>> from sympy.abc import a, b, x, y, z
>>> from sympy import Symbol, exp, log, sqrt, symbols, powdenest
```

```
>>> powdenest((x**(2*a/3))**(3*x))
(x**(2*a/3))**(3*x)
>>> powdenest(exp(3*x*log(2)))
2**(3*x)
```

Assumptions may prevent expansion:

```
>>> powdenest(sqrt(x**2))
sqrt(x**2)
```

```
>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p
```

No other expansion is done.

```
>>> i, j = symbols('i,j', integer=True)
>>> powdenest((x**x)**(i + j)) # -X-> (x**x)**i*(x**x)**j
x**(x*(i + j))
```

But `exp()` will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the `exp` to a power with a different base:

```
>>> powdenest(exp(3*y*log(x)))
x**(3*y)
>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y
>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3
```

If assumptions allow, symbols can also be moved to the outermost exponent:

```
>>> i = Symbol('i', integer=True)
>>> powdenest(((x**(2*i))**(3*y))**x)
((x**(2*i))**(3*y))**x
>>> powdenest(((x**(2*i))**(3*y))**x, force=True)
x**(6*i*x*y)
```

```
>>> powdenest(((x**(2*a/3))**(3*y/i))**x)
((x**(2*a/3))**(3*y/i))**x
>>> powdenest((x**(2*i)*y**(4*i))**z, force=True)
(x*y**2)**(2*i*z)
```

```
>>> n = Symbol('n', negative=True)
```

```
>>> powdenest((x**i)**y, force=True)
x**(i*y)
>>> powdenest((n**i)**x, force=True)
(n**i)**x
```

5.26.14 Combinatorial simplification

combsimp

`sympy.simplify.combsimp.combsimp(expr)`

Simplify combinatorial expressions.

This function takes as input an expression containing factorials, binomials, Pochhammer symbol and other “combinatorial” functions, and tries to minimize the number of those functions and reduce the size of their arguments.

The algorithm works by rewriting all combinatorial functions as gamma functions and applying `gammasimp()` except simplification steps that may make an integer argument non-integer. See docstring of `gammasimp` for more information.

Then it rewrites expression in terms of factorials and binomials by rewriting gammas as factorials and converting $(a+b)!/a!b!$ into binomials.

If expression has gamma functions or combinatorial functions with non-integer argument, it is automatically passed to `gammasimp`.

Examples

```
>>> from sympy.simplify import combsimp
>>> from sympy import factorial, binomial, symbols
>>> n, k = symbols('n k', integer = True)
```

```
>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
(n + 1)/(k + 1)
```

5.26.15 Square Root Denesting

sqrtdenest

`sympy.simplify.sqrtdenest.sqrtdenest(expr, max_iter=3)`

Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged. This is based on the algorithms of [1].

See also:

`sympy.solvers.solvers.unrad`

References

- [1] <http://researcher.watson.ibm.com/researcher/files/us-fagin/symb85.pdf>
- [2] D. J. Jeffrey and A. D. Rich, 'Simplifying Square Roots of Square Roots by Denesting' (available at <http://www.cybertester.com/data/denest.pdf>)

Examples

```
>>> from sympy.simplify.sqrtdenest import sqrtdenest
>>> from sympy import sqrt
>>> sqrtdenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

5.26.16 Common Subexpression Elimination

cse

`sympy.simplify.cse_main.cse`(*exprs*, *symbols*=None, *optimizations*=None, *postprocess*=None, *order*='canonical', *ignore*=())

Perform common subexpression elimination on an expression.

Parameters *exprs* : list of sympy expressions, or a single sympy expression

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out. The `numbered_symbols` generator is useful. The default is a stream of symbols of the form "x0", "x1", etc. This must be an infinite iterator.

optimizations : list of (callable, callable) pairs

The (preprocessor, postprocessor) pairs of external optimization functions. Optionally 'basic' can be passed for a set of predefined basic optimizations. Such 'basic' optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

postprocess : a function which accepts the two return values of cse and returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda:
lambda r, e: return reversed(r), e

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. If set to 'canonical', arguments will be canonically ordered. If set to 'none', ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting `order='none'`.

ignore : iterable of Symbols

Substitutions containing any Symbol from ignore will be ignored.

Returns replacements : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

reduced_exprs : list of sympy expressions

The reduced expressions with all of the replacements above.

Examples

```
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

Note that currently, $y + z$ will not get substituted if $-y - z$ is used.

```
>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols: `>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))` `([(x0, x + 1)], [x0*y**2, 3*x0*y**2])`

opt_cse

`sympy.simplify.cse_main.opt_cse(exprs, order='canonical')`

Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls

Parameters exprs : list of sympy expressions

The expressions to optimize.

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.

Returns opt_subs : dictionary of expression substitutions

The expression substitutions which can be useful to optimize CSE.

Examples

```
>>> from sympy.simplify.cse_main import opt_cse
>>> from sympy.abc import x
>>> opt_subs = opt_cse([x**(-2)])
>>> k, v = list(opt_subs.keys())[0], list(opt_subs.values())[0]
>>> print((k, v.as_unevaluated_basic()))
(x**(-2), 1/(x**2))
```

tree_cse

`sympy.simplify.cse_main.tree_cse(exprs, symbols, opt_subs=None, or-
der='canonical', ignore=())`
Perform raw CSE on expression tree, taking `opt_subs` into account.

Parameters **exprs** : list of sympy expressions

The expressions to reduce.

symbols : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out.

opt_subs : dictionary of expression substitutions

The expressions to be substituted before any CSE action is performed.

order : string, 'none' or 'canonical'

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.

ignore : iterable of Symbols

Substitutions containing any Symbol from `ignore` will be ignored.

5.26.17 Hypergeometric Function Expansion

hyperexpand

`sympy.simplify.hyperexpand.hyperexpand(f, allow_hyper=False, rewrite='default',
place=None)`

Expand hypergeometric functions. If `allow_hyper` is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

If a G-function has expansions both at zero and at infinity, `place` can be set to 0 or zoo to indicate the preferred choice.

Examples

```
>>> from sympy.simplify.hyperexpand import hyperexpand
>>> from sympy.functions import hyper
>>> from sympy.abc import z
```

(continues on next page)

(continued from previous page)

```
>>> hyperexpand(hyper([], [], z))
exp(z)
```

Non-hypergeometric parts of the expression and hypergeometric expressions that are not recognised are left unchanged:

```
>>> hyperexpand(1 + hyper([1, 1, 1], [], z))
hyper((1, 1, 1), (), z) + 1
```

5.26.18 Traversal Tools

use

`sympy.simplify.traversaltools.use(expr, func, level=0, args=(), kwargs={})`
 Use func to transform expr at the given level.

Examples

```
>>> from sympy import use, expand
>>> from sympy.abc import x, y
```

```
>>> f = (x + y)**2*x + 1
```

```
>>> use(f, expand, level=2)
x*(x**2 + 2*x*y + y**2) + 1
>>> expand(f)
x**3 + 2*x**2*y + x*y**2 + 1
```

5.26.19 EPath Tools

EPath class

class `sympy.simplify.epathtools.EPath`

Manipulate expressions using paths.

EPath grammar in EBNF notation:

```
literal  ::= /[A-Za-z_][A-Za-z_0-9]*/
number   ::= /-?\d+/
type     ::= literal
attribute ::= literal "?"
all       ::= "*"
slice     ::= "[" number? (":" number? (":" number?))?" "]"
range     ::= all | slice
query     ::= (type | attribute) ("|" (type | attribute))*
selector  ::= range | query range?
path      ::= "/" selector ("/" selector)*
```

See the docstring of the `epath()` function.

apply(expr, func, args=None, kwargs=None)
Modify parts of an expression selected by a path.

Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> path.apply(expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.apply(expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

select(expr)
Retrieve parts of an expression selected by a path.

Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> path.select(expr)
[x, y]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.select(expr)
[x, x, y]
```

epath

sympy.simplify.epathtools.epath(path, expr=None, func=None, args=None, kwargs=None)

Manipulate parts of an expression selected by a path.

This function allows to manipulate large nested expressions in single line of code, utilizing techniques to those applied in XML processing standards (e.g. XPath).

If `func` is `None`, `epath()` (page 1178) retrieves elements selected by the path. Otherwise it applies `func` to each matching element.

Note that it is more efficient to create an `EPath` object and use the `select` and `apply` methods of that object, since this will compile the path string only once. This function should only be used as a convenient shortcut for interactive use.

This is the supported syntax:

- **select all:** `/*` Equivalent of `for arg in args:`.
- **select slice:** `/[0]` or `/[1:5]` or `/[1:5:2]` Supports standard Python's slice syntax.
- **select by type:** `/list` or `/list|tuple` Emulates `isinstance()`.
- **select by attribute:** `/__iter__?` Emulates `hasattr()`.

Parameters `path` : `str` | `EPath`

A path as a string or a compiled `EPath`.

expr : `Basic` | `iterable`

An expression or a container of expressions.

func : callable (optional)

A callable that will be applied to matching parts.

args : `tuple` (optional)

Additional positional arguments to `func`.

kwargs : `dict` (optional)

Additional keyword arguments to `func`.

Examples

```
>>> from sympy.simplify.epathtools import epath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = "/*/[0]/Symbol"
>>> expr = [(x, 1), 2], ((3, y), z)]
```

```
>>> epath(path, expr)
[x, y]
>>> epath(path, expr, lambda expr: expr**2)
[(x**2, 1), 2], ((3, y**2), z)]
```

```
>>> path = "/*/*Symbol"
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> epath(path, expr)
[x, x, y]
>>> epath(path, expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

5.27 Details on the Hypergeometric Function Expansion Module

This page describes how the function `hyperexpand()` and related code work. For usage, see the documentation of the `simplify` module.

5.27.1 Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roach1996] (page 1869) and [Roach1997] (page 1869).

Recall that the hypergeometric function is (initially) defined as

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}.$$

It turns out that there are certain differential operators that can change the a_p and p_q parameters by integers. If a sequence of such operators is known that converts the set of indices a_r^0 and b_s^0 into a_p and b_q , then we shall say the pair a_p, b_q is reachable from a_r^0, b_s^0 . Our general strategy is thus as follows: given a set a_p, b_q of parameters, try to look up an origin a_r^0, b_s^0 for which we know an expression, and then apply the sequence of differential operators to the known expression to find an expression for the Hypergeometric function we are interested in.

Notation

In the following, the symbol a will always denote a numerator parameter and the symbol b will always denote a denominator parameter. The subscripts p, q, r, s denote vectors of that length, so e.g. a_p denotes a vector of p numerator parameters. The subscripts i and j denote “running indices”, so they should usually be used in conjunction with a “for all i ”. E.g. $a_i < 4$ for all i . Uppercase subscripts I and J denote a chosen, fixed index. So for example $a_I > 0$ is true if the inequality holds for the one index I we are currently interested in.

Incrementing and decrementing indices

Suppose $a_i \neq 0$. Set $A(a_i) = \frac{z}{a_i} \frac{d}{dz} + 1$. It is then easy to show that $A(a_i) {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p + e_i \\ b_q \end{matrix} \middle| z \right)$, where e_i is the i -th unit vector. Similarly for $b_j \neq 1$ we set $B(b_j) = \frac{z}{b_j - 1} \frac{d}{dz} + 1$ and find $B(b_j) {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p \\ b_q - e_j \end{matrix} \middle| z \right)$. Thus we can increment upper and decrement lower indices at will, as long as we don’t go through zero. The $A(a_i)$ and $B(b_j)$ are called shift operators.

It is also easy to show that $\frac{d}{dz} {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = \frac{a_1 \cdots a_p}{b_1 \cdots b_q} {}_pF_q \left(\begin{matrix} a_p + 1 \\ b_q + 1 \end{matrix} \middle| z \right)$, where $a_p + 1$ is the vector $a_1 + 1, a_2 + 1, \dots$ and similarly for $b_q + 1$. Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: $\left[\frac{d}{dz} \prod_{j=1}^q B(b_j) - \frac{a_1 \cdots a_p}{(b_1 - 1) \cdots (b_q - 1)} \prod_{i=1}^p A(a_i) \right] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$. This holds if all shift operators are defined, i.e. if no $a_i = 0$ and no $b_j = 1$. Clearing denominators and multiplying through by z we arrive at the following equation: $\left[z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) - z \prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) \right] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$. Even though our derivation does not show it, it can be checked that this equation holds whenever the ${}_pF_q$ is defined.

Notice that, under suitable conditions on a_I, b_J , each of the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ can be expressed in terms of $A(a_I)$ or $B(b_J)$. Our next aim is to write the Hypergeometric differential equation as follows: $[XA(a_I) - r] {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$, for some operator X and some constant r to be determined. If $r \neq 0$, then we can write this as $\frac{-1}{r} X {}_pF_q \left(\begin{smallmatrix} a_p + e_I \\ b_q \end{smallmatrix} \middle| z \right) = {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right)$, and so $\frac{-1}{r} X$ undoes the shifting of $A(a_I)$, whence it will be called an inverse-shift operator.

Now $A(a_I)$ exists if $a_I \neq 0$, and then $z \frac{d}{dz} = a_I A(a_I) - a_I$. Observe also that all the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ commute. We have $\prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) = \left(\prod_{i=1, i \neq I}^p \left(z \frac{d}{dz} + a_i \right) \right) a_I A(a_I)$, so this gives us the first half of X . The other half does not have such a nice expression. We find $z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) = (a_I A(a_I) - a_I) \prod_{j=1}^q (a_I A(a_I) - a_I + b_j - 1)$. Since the first half had no constant term, we infer $r = -a_I \prod_{j=1}^q (b_j - 1 - a_I)$.

This tells us under which conditions we can “un-shift” $A(a_I)$, namely when $a_I \neq 0$ and $r \neq 0$. Substituting $a_I - 1$ for a_I then tells us under what conditions we can decrement the index a_I . Doing a similar analysis for $B(b_J)$, we arrive at the following rules:

- An index a_I can be decremented if $a_I \neq 1$ and $a_I \neq b_j$ for all b_j .
- An index b_J can be incremented if $b_J \neq -1$ and $b_J \neq a_i$ for all a_i .

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

Reduction of Order

Notice that, quite trivially, if $a_I = b_J$, we have ${}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_{p-1}F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$, where a_p^* means a_p with a_I omitted, and similarly for b_q^* . We call this reduction of order.

In fact, we can do even better. If $a_I - b_J \in \mathbb{Z}_{>0}$, then it is easy to see that $\frac{(a_I)_n}{(b_J)_n}$ is actually a polynomial in n . It is also easy to see that $(z \frac{d}{dz})^k z^n = n^k z^n$. Combining these two remarks we find:

If $a_I - b_J \in \mathbb{Z}_{>0}$, then there exists a polynomial $p(n) = p_0 + p_1 n + \dots$ (of degree $a_I - b_J$) such that $\frac{(a_I)_n}{(b_J)_n} = p(n)$ and ${}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = \left(p_0 + p_1 z \frac{d}{dz} + p_2 \left(z \frac{d}{dz} \right)^2 + \dots \right) {}_{p-1}F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$.

Thus any set of parameters a_p, b_q is reachable from a set of parameters c_r, d_s where $c_i - d_j \in \mathbb{Z}$ implies $c_i < d_j$. Such a set of parameters c_r, d_s is called suitable. Our database of known formulae should only contain suitable origins. The reasons are twofold: firstly, working from suitable origins is easier, and secondly, a formula for a non-suitable origin can be deduced from a lower order formula, and we should put this one into the database instead.

Moving Around in the Parameter Space

It remains to investigate the following question: suppose a_p, b_q and a_p^0, b_q^0 are both suitable, and also $a_i - a_i^0 \in \mathbb{Z}$, $b_j - b_j^0 \in \mathbb{Z}$. When is a_p, b_q reachable from a_p^0, b_q^0 ? It is clear that we can treat all parameters independently that are incongruent mod 1. So assume that a_i and b_j are congruent to r mod 1, for all i and j . The same then follows for a_i^0 and b_j^0 .

If $r \neq 0$, then any such a_p, b_q is reachable from any a_p^0, b_q^0 . To see this notice that there exist constants c, c^0 , congruent mod 1, such that $a_i < c < b_j$ for all i and j , and similarly $a_i^0 < c^0 < b_j^0$. If $n = c - c^0 > 0$ then we first inverse-shift all the b_j^0 n times up, and then similarly shift up all the a_i^0 n times. If $n < 0$ then we first inverse-shift down the a_i^0 and then shift down the b_j^0 .

This reduces to the case $c = c^0$. But evidently we can now shift or inverse-shift around the a_i^0 arbitrarily so long as we keep them less than c , and similarly for the b_j^0 so long as we keep them bigger than c . Thus a_p, b_q is reachable from a_p^0, b_q^0 .

If $r = 0$ then the problem is slightly more involved. WLOG no parameter is zero. We now have one additional complication: no parameter can ever move through zero. Hence a_p, b_q is reachable from a_p^0, b_q^0 if and only if the number of $a_i < 0$ equals the number of $a_i^0 < 0$, and similarly for the b_i and b_i^0 . But in a suitable set of parameters, all $b_j > 0$! This is because the Hypergeometric function is undefined if one of the b_j is a non-positive integer and all a_i are smaller than the b_j . Hence the number of $b_j \leq 0$ is always zero.

We can thus associate to every suitable set of parameters a_p, b_q , where no $a_i = 0$, the following invariants:

- For every $r \in [0, 1)$ the number α_r of parameters $a_i \equiv r \pmod{1}$, and similarly the number β_r of parameters $b_i \equiv r \pmod{1}$.
- The number γ of integers a_i with $a_i < 0$.

The above reasoning shows that a_p, b_q is reachable from a_p^0, b_q^0 if and only if the invariants $\alpha_r, \beta_r, \gamma$ all agree. Thus in particular “being reachable from” is a symmetric relation on suitable parameters without zeros.

Applying the Operators

If all goes well then for a given set of parameters we find an origin in our database for which we have a nice formula. We now have to apply (potentially) many differential operators to it. If we do this blindly then the result will be very messy. This is because with Hypergeometric type functions, the derivative is usually expressed as a sum of two contiguous functions. Hence if we compute N derivatives, then the answer will involve $2N$ contiguous functions! This is clearly undesirable. In fact we know from the Hypergeometric differential equation that we need at most $\max(p, q + 1)$ contiguous functions to express all derivatives.

Hence instead of differentiating blindly, we will work with a $\mathbb{C}(z)$ -module basis: for an origin a_r^0, b_s^0 we either store (for particularly pretty answers) or compute a set of N functions (typically $N = \max(r, s + 1)$) with the property that the derivative of any of them is a $\mathbb{C}(z)$ -linear combination of them. In formulae, we store a vector B of N functions, a matrix M and a vector C (the latter two with entries in $\mathbb{C}(z)$), with the following properties:

- ${}_rF_s \left(\begin{smallmatrix} a_r^0 \\ b_s^0 \end{smallmatrix} \middle| z \right) = CB$
- $z \frac{d}{dz} B = MB$.

Then we can compute as many derivatives as we want and we will always end up with $\mathbb{C}(z)$ -linear combination of at most N special functions.

As hinted above, B , M and C can either all be stored (for particularly pretty answers) or computed from a single ${}_pF_q$ formula.

Loose Ends

This describes the bulk of the hypergeometric function algorithm. There a few further tricks, described in the `hyperexpand.py` source file. The extension to Meijer G-functions is also described there.

5.27.2 Meijer G-Functions of Finite Confluence

Slater's theorem essentially evaluates a G -function as a sum of residues. If all poles are simple, the resulting series can be recognised as hypergeometric series. Thus a G -function can be evaluated as a sum of Hypergeometric functions.

If the poles are not simple, the resulting series are not hypergeometric. This is known as the "confluent" or "logarithmic" case (the latter because the resulting series tend to contain logarithms). The answer depends in a complicated way on the multiplicities of various poles, and there is no accepted notation for representing it (as far as I know). However if there are only finitely many multiple poles, we can evaluate the G function as a sum of hypergeometric functions, plus finitely many extra terms. I could not find any good reference for this, which is why I work it out here.

Recall the general setup. We define

$$G(z) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where L is a contour starting and ending at $+\infty$, enclosing all of the poles of $\Gamma(b_j - s)$ for $j = 1, \dots, n$ once in the negative direction, and no other poles. Also the integral is assumed absolutely convergent.

In what follows, for any complex numbers a, b , we write $a \equiv b \pmod{1}$ if and only if there exists an integer k such that $a - b = k$. Thus there are double poles iff $a_i \equiv a_j \pmod{1}$ for some $i \neq j \leq n$.

We now assume that whenever $b_j \equiv a_i \pmod{1}$ for $i \leq m$, $j > n$ then $b_j < a_i$. This means that no quotient of the relevant gamma functions is a polynomial, and can always be achieved by "reduction of order". Fix a complex number c such that $\{b_i | b_i \equiv c \pmod{1}, i \leq m\}$ is not empty. Enumerate this set as $b, b + k_1, \dots, b + k_u$, with k_i non-negative integers. Enumerate similarly $\{a_j | a_j \equiv c \pmod{1}, j > n\}$ as $b + l_1, \dots, b + l_v$. Then $l_i > k_j$ for all i, j . For finite confluence, we need to assume $v \geq u$ for all such c .

Let $c_1, \dots, c_w$ be distinct $\pmod{1}$ and exhaust the congruence classes of the b_i . I claim

$$G(z) = - \sum_{j=1}^w (F_j(z) + R_j(z)),$$

where $F_j(z)$ is a hypergeometric function and $R_j(z)$ is a finite sum, both to be specified later. Indeed corresponding to every c_j there is a sequence of poles, at mostly finitely many of them multiple poles. This is where the j -th term comes from.

Hence fix again c , enumerate the relevant b_i as $b, b + k_1, \dots, b + k_u$. We will look at the a_j corresponding to $a + l_1, \dots, a + l_u$. The other a_i are not treated specially. The corresponding gamma functions have poles at (potentially) $s = b + r$ for $r = 0, 1, \dots$. For $r \geq l_u$, pole of the integrand is simple. We thus set

$$R(z) = \sum_{r=0}^{l_u-1} \text{res}_{s=r+b}.$$

We finally need to investigate the other poles. Set $r = l_u + t$, $t \geq 0$. A computation shows

$$\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)_{l_i - k_i}} = \frac{(-1)^{\delta_i}}{(l_u - l_i + 1)_{\delta_i}} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t},$$

where $\delta_i = l_i - k_i$.

Also

$$\Gamma(b_j - l_u - b - t) = \frac{\Gamma(b_j - l_u - b)}{(-1)^t (l_u + b + 1 - b_j)_t},$$

$$\Gamma(1 - a_j + l_u + b + t) = \Gamma(1 - a_j + l_u + b)(1 - a_j + l_u + b)_t$$

and

$$res_{s=b+l_u+t} \Gamma(b - s) = -\frac{(-1)^{l_u+t}}{(l_u + t)!} = -\frac{(-1)^{l_u}}{l_u!} \frac{(-1)^t}{(l_u + 1)_t}.$$

Hence

$$\begin{aligned} res_{s=b+l_u+t} &= -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^u \frac{(-1)^{\delta_i}}{(l_u - k_i + 1)_{\delta_i}} \frac{\prod_{j=1}^n \Gamma(1 - a_j + l_u + b) \prod_{j=1}^m \Gamma(b_j - l_u - b)^*}{\prod_{j=n+1}^p \Gamma(a_j - l_u - b)^* \prod_{j=m+1}^q \Gamma(1 - b_j + l_u + b)} \\ &\times z^t \frac{(-1)^t}{(l_u + 1)_t} \prod_{i=1}^u \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t} \frac{\prod_{j=1}^n (1 - a_j + l_u + b)_t \prod_{j=n+1}^p (-1)^t (l_u + b + 1 - a_j)_t^*}{\prod_{j=1}^m (-1)^t (l_u + b + 1 - b_j)_t^* \prod_{j=m+1}^q (1 - b_j + l_u + b)_t}, \end{aligned}$$

where the * means to omit the terms we treated specially.

We thus arrive at

$$F(z) = C \times {}_{p+1}F_q \left(\begin{matrix} 1, (1 + l_u - l_i), (1 + l_u + b - a_i)^* \\ 1 + l_u, (1 + l_u - k_i), (1 + l_u + b - b_i)^* \end{matrix} \middle| (-1)^{p-m-n} z \right),$$

where C designates the factor in the residue independent of t . (This result can also be written in slightly simpler form by converting all the l_u etc back to a_* - b_* , but doing so is going to require more notation still and is not helpful for computation.)

5.27.3 Extending The Hypergeometric Tables

Adding new formulae to the tables is straightforward. At the top of the file `sympy/simplify/hyperexpand.py`, there is a function called `add_formulae()`. Nested in it are defined two helpers, `add(ap, bq, res)` and `addb(ap, bq, B, C, M)`, as well as dummies `a`, `b`, `c`, and `z`.

The first step in adding a new formula is by using `add(ap, bq, res)`. This declares `hyper(ap, bq, z) == res`. Here `ap` and `bq` may use the dummies `a`, `b`, and `c` as free symbols. For example the well-known formula $\sum_0^\infty \frac{(-a)_n z^n}{n!} = (1 - z)^a$ is declared by the following line: `add((-a, ), (), (1-z)**a)`.

From the information provided, the matrices B , C and M will be computed, and the formula is now available when expanding hypergeometric functions. Next the test file `sympy/simplify/tests/test_hyperexpand.py` should be run, in particular the test `test_formulae()`. This will test the newly added formula numerically. If it fails, there is (presumably) a typo in what was entered.

Since all newly-added formulae are probably relatively complicated, chances are that the automatically computed basis is rather suboptimal (there is no good way of testing this, other than observing very messy output). In this case the matrices B , C and M should be computed by hand. Then the helper `addb` can be used to declare a hypergeometric formula with hand-computed basis.

An example

Because this explanation so far might be very theoretical and difficult to understand, we walk through an explicit example now. We take the Fresnel function $C(z)$ which obeys the following hypergeometric representation:

$$C(z) = z \cdot {}_1F_2 \left(\frac{1}{2} \middle| \frac{5}{4} - \frac{\pi^2 z^4}{16} \right).$$

First we try to add this formula to the lookup table by using the (simpler) function `add(ap, bq, res)`. The first two arguments are simply the lists containing the parameter sets of ${}_1F_2$. The `res` argument is a little bit more complicated. We only know $C(z)$ in terms of ${}_1F_2(\dots|f(z))$ with f a function of z , in our case

$$f(z) = -\frac{\pi^2 z^4}{16}.$$

What we need is a formula where the hypergeometric function has only z as argument ${}_1F_2(\dots|z)$. We introduce the new complex symbol w and search for a function $g(w)$ such that

$$f(g(w)) = w$$

holds. Then we can replace every z in $C(z)$ by $g(w)$. In the case of our example the function g could look like

$$g(w) = \frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}.$$

We get these functions mainly by guessing and testing the result. Hence we proceed by computing $f(g(w))$ (and simplifying naively)

$$\begin{aligned} f(g(w)) &= -\frac{\pi^2 g(w)^4}{16} \\ &= -\frac{\pi^2 g\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)^4}{16} \\ &= -\frac{\pi^2 \frac{2^4}{\sqrt{\pi}^4} \exp\left(\frac{i\pi}{4}\right)^4 w^{\frac{1}{4}^4}}{16} \\ &= -\exp(i\pi) w \\ &= w \end{aligned}$$

and indeed get back w . (In case of branched functions we have to be aware of branch cuts. In that case we take w to be a positive real number and check the formula. If what we have found works for positive w , then just replace `exp()` inside any branched function by `exp_polar()` and what we get is right for *all* w .) Hence we can write the formula as

$$C(g(w)) = g(w) \cdot {}_1F_2 \left(\frac{1}{2} \middle| \frac{5}{4} w \right).$$

and trivially

$${}_1F_2 \left(\frac{1}{2} \middle| \frac{5}{4} w \right) = \frac{C(g(w))}{g(w)} = \frac{C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)}{\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third parameter, `res`, in `add`. Finally, the whole function call to add this rule to the table looks like:

```
add([S(1)/4],
    [S(1)/2, S(5)/4],
    fresnelc(exp(pi*I/4)*root(z,4)*2/sqrt(pi)) / (exp(pi*I/4)*root(z,4)*2/sqrt(pi))
)
```

Using this rule we will find that it works but the results are not really nice in terms of simplicity and number of special function instances included. We can obtain much better results by adding the formula to the lookup table in another way. For this we use the (more complicated) function `adddb(ap, bq, B, C, M)`. The first two arguments are again the lists containing the parameter sets of ${}_1F_2$. The remaining three are the matrices mentioned earlier on this page.

We know that the $n = \max(p, q + 1)$ -th derivative can be expressed as a linear combination of lower order derivatives. The matrix B contains the basis $\{B_0, B_1, \dots\}$ and is of shape $n \times 1$. The best way to get B_i is to take the first $n = \max(p, q + 1)$ derivatives of the expression for ${}_pF_q$ and take out useful pieces. In our case we find that $n = \max(1, 2 + 1) = 3$. For computing the derivatives, we have to use the operator $z \frac{d}{dz}$. The first basis element B_0 is set to the expression for ${}_1F_2$ from above:

$$B_0 = \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}}$$

Next we compute $z \frac{d}{dz} B_0$. For this we can directly use SymPy!

```
>>> from sympy import Symbol, sqrt, exp, I, pi, fresnelc, root, diff, expand
>>> z = Symbol("z")
>>> B0 = sqrt(pi)*exp(-I*pi/4)*fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/\
...     (2*root(z,4))
>>> z * diff(B0, z)
z*(cosh(2*sqrt(z))/(4*z) - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/\
↳ sqrt(pi))/(8*z**(5/4)))
>>> expand(_)
cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/\
↳ (8*z**(1/4))
```

Formatting this result nicely we obtain

$$B'_1 = -\frac{1}{4} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} + \frac{1}{4} \cosh(2\sqrt{z})$$

Computing the second derivative we find

```
>>> from sympy import (Symbol, cosh, sqrt, pi, exp, I, fresnelc, root,
...                     diff, expand)
>>> z = Symbol("z")
>>> B1prime = cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*\
...     fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/(8*root(z,4))
>>> z * diff(B1prime, z)
z*(-cosh(2*sqrt(z))/(16*z) + sinh(2*sqrt(z))/(4*sqrt(z)) + sqrt(pi)*exp(-I*pi/\
↳ 4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/(32*z**(5/4)))
>>> expand(_)
sqrt(z)*sinh(2*sqrt(z))/4 - cosh(2*sqrt(z))/16 + sqrt(pi)*exp(-I*pi/\
↳ 4)*fresnelc(2*z**(1/4)*exp(I*pi/4)/sqrt(pi))/(32*z**(1/4))
```

which can be printed as

$$B'_2 = \frac{1}{16} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16} \cosh(2\sqrt{z}) + \frac{1}{4} \sinh(2\sqrt{z})\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose B_1 and B_2 as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{\pi} \exp(-\frac{i\pi}{4}) C(\frac{2}{\sqrt{\pi}} \exp(\frac{i\pi}{4}) z^{\frac{1}{4}})}{2z^{\frac{1}{4}} \cosh(2\sqrt{z})} \\ \cosh(2\sqrt{z}) \\ \sinh(2\sqrt{z}) \sqrt{z} \end{pmatrix}$$

(This is in contrast to the basis $B = (B_0, B'_1, B'_2)$ that would have been computed automatically if we used just `add(ap, bq, res)`.)

Because it must hold that ${}_pF_q(\cdots|z) = CB$ the entries of C are obviously

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Finally we have to compute the entries of the 3×3 matrix M such that $z \frac{d}{dz} B = MB$ holds. This is easy. We already computed the first part $z \frac{d}{dz} B_0$ above. This gives us the first row of M . For the second row we have:

```
>>> from sympy import Symbol, cosh, sqrt, diff
>>> z = Symbol("z")
>>> B1 = cosh(2*sqrt(z))
>>> z * diff(B1, z)
sqrt(z)*sinh(2*sqrt(z))
```

and for the third one

```
>>> from sympy import Symbol, sinh, sqrt, expand, diff
>>> z = Symbol("z")
>>> B2 = sinh(2*sqrt(z))*sqrt(z)
>>> expand(z * diff(B2, z))
sqrt(z)*sinh(2*sqrt(z))/2 + z*cosh(2*sqrt(z))
```

Now we have computed the entries of this matrix to be

$$M = \begin{pmatrix} -\frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 1 \\ 0 & z & \frac{1}{2} \end{pmatrix}$$

Note that the entries of C and M should typically be rational functions in z , with rational coefficients. This is all we need to do in order to add a new formula to the lookup table for `hyperexpand`.

5.27.4 Implemented Hypergeometric Formulae

A vital part of the algorithm is a relatively large table of hypergeometric function representations. The following automatically generated list contains all the representations implemented in SymPy (of course many more are derived from them). These formulae are mostly taken from [Luke1969] (page 1869) and [Prudnikov1990] (page 1869). They are all tested numerically.

$${}_0F_0(|z) = e^z$$

$$\begin{aligned}
{}_1F_0(a|z) &= (-z+1)^{-a} \\
{}_2F_1\left(a, a - \frac{1}{2} \middle| z\right) &= 2^{2a-1} (\sqrt{-z+1} + 1)^{-2a+1} \\
{}_2F_1\left(\frac{1}{2}, 1 \middle| z\right) &= -\frac{\log(-z+1)}{z} \\
{}_2F_1\left(\frac{1}{2}, \frac{3}{2} \middle| z\right) &= \frac{\operatorname{atanh}(\sqrt{z})}{\sqrt{z}} \\
{}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{\operatorname{asin}(\sqrt{z})}{\sqrt{z}} \\
{}_2F_1\left(a, a + \frac{1}{2} \middle| z\right) &= \frac{(\sqrt{z}+1)^{-2a}}{2} + \frac{(-\sqrt{z}+1)^{-2a}}{2} \\
{}_2F_1\left(a, -\frac{1}{2} \middle| z\right) &= \cos(2a \operatorname{asin}(\sqrt{z})) \\
{}_2F_1\left(1, 1 \middle| z\right) &= \frac{\operatorname{asin}(\sqrt{z})}{\sqrt{z}\sqrt{-z+1}} \\
{}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{2K(z)}{\pi} \\
{}_2F_1\left(-\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{2E(z)}{\pi} \\
{}_3F_2\left(-\frac{1}{2}, 1, 1 \middle| \frac{1}{2}, 2 \middle| z\right) &= -\frac{2\sqrt{z} \operatorname{atanh}(\sqrt{z})}{3} + \frac{2}{3} - \frac{\log(-z+1)}{3z} \\
{}_3F_2\left(-\frac{1}{2}, 1, 1 \middle| \frac{1}{2}, 2 \middle| z\right) &= \left(\frac{4}{9} - \frac{16}{9z}\right) \sqrt{-z+1} + \frac{4 \log\left(\frac{\sqrt{-z+1}}{2} + \frac{1}{2}\right)}{3z} + \frac{16}{9z} \\
{}_1F_1\left(\frac{1}{b} \middle| z\right) &= z^{-b+1} (b-1) e^z \gamma(b-1, z) \\
{}_1F_1\left(\frac{a}{2a} \middle| z\right) &= 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} e^{\frac{z}{2}} I_{a-\frac{1}{2}}\left(\frac{z}{2}\right) \Gamma\left(a + \frac{1}{2}\right) \\
{}_1F_1\left(\frac{a}{a+1} \middle| z\right) &= a (ze^{i\pi})^{-a} \gamma(a, ze^{i\pi}) \\
{}_1F_1\left(-\frac{1}{2} \middle| z\right) &= \sqrt{z} i \sqrt{\pi} \operatorname{erf}(\sqrt{z} i) + e^z \\
{}_1F_2\left(\frac{1}{\frac{3}{4}, \frac{5}{4}} \middle| z\right) &= \frac{\sqrt{\pi} \left(i \sinh(2\sqrt{z}) S\left(\frac{2\sqrt[4]{ze^{\frac{i\pi}{4}}}}{\sqrt{\pi}}\right) + \cosh(2\sqrt{z}) C\left(\frac{2\sqrt[4]{ze^{\frac{i\pi}{4}}}}{\sqrt{\pi}}\right) \right)}{2\sqrt[4]{z}} e^{-\frac{i\pi}{4}} \\
{}_2F_2\left(\frac{1}{2}, a \middle| \frac{3}{2}, a+1 \middle| z\right) &= -\frac{ai\sqrt{\pi}\sqrt{\frac{1}{z}} \operatorname{erf}(\sqrt{z} i)}{2a-1} - \frac{a (ze^{i\pi})^{-a} \gamma(a, ze^{i\pi})}{2a-1} \\
{}_2F_2\left(1, 1 \middle| 2, 2 \middle| z\right) &= \frac{-\log(z) + \operatorname{Ei}(z)}{z} - \frac{\gamma}{z} \\
{}_0F_1\left(\frac{1}{2} \middle| z\right) &= \cosh(2\sqrt{z}) \\
{}_0F_1\left(b \middle| z\right) &= z^{-\frac{b}{2}+\frac{1}{2}} I_{b-1}(2\sqrt{z}) \Gamma(b)
\end{aligned}$$

$$\begin{aligned}
{}_0F_3\left(\frac{1}{2}, a, a + \frac{1}{2} \middle| z\right) &= 2^{-2a} z^{-\frac{a}{2} + \frac{1}{4}} \left(I_{2a-1} \left(4\sqrt[4]{z} \right) + J_{2a-1} \left(4\sqrt[4]{z} \right) \right) \Gamma(2a) \\
{}_0F_3\left(a, a + \frac{1}{2}, 2a \middle| z\right) &= \left(2\sqrt{z} e^{\frac{i\pi}{2}} \right)^{-2a+1} I_{2a-1} \left(2\sqrt{2}\sqrt[4]{z} e^{\frac{i\pi}{4}} \right) J_{2a-1} \left(2\sqrt{2}\sqrt[4]{z} e^{\frac{i\pi}{4}} \right) \Gamma^2(2a) \\
{}_1F_2\left(a - \frac{1}{2}, 2a \middle| z\right) &= 2 \cdot 4^{a-1} z^{-a+1} I_{a-\frac{3}{2}}(\sqrt{z}) I_{a-\frac{1}{2}}(\sqrt{z}) \Gamma\left(a - \frac{1}{2}\right) \Gamma\left(a + \frac{1}{2}\right) - 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} I_{a-\frac{1}{2}}^2(\sqrt{z}) \Gamma^2\left(a + \frac{1}{2}\right) \\
{}_1F_2\left(b, -b + 2 \middle| z\right) &= \frac{\pi(-b+1) I_{-b+1}(\sqrt{z}) I_{b-1}(\sqrt{z})}{\sin(b\pi)} \\
{}_1F_2\left(\frac{1}{2}, \frac{3}{2} \middle| z\right) &= \frac{\operatorname{Shi}(2\sqrt{z})}{2\sqrt{z}} \\
{}_1F_2\left(\frac{3}{2}, \frac{7}{4} \middle| z\right) &= \frac{3\sqrt{\pi} e^{-\frac{3i\pi}{4}} S\left(\frac{2\sqrt[4]{z} e^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right)}{4z^{\frac{3}{4}}} \\
{}_1F_2\left(\frac{1}{2}, \frac{5}{4} \middle| z\right) &= \frac{\sqrt{\pi} e^{-\frac{i\pi}{4}} C\left(\frac{2\sqrt[4]{z} e^{\frac{i\pi}{4}}}{\sqrt{\pi}}\right)}{2\sqrt[4]{z}} \\
{}_2F_3\left(a, a + \frac{1}{2}, 2a, b, 2a - b + 1 \middle| z\right) &= \left(\frac{\sqrt{z}}{2}\right)^{-2a+1} I_{2a-b}(\sqrt{z}) I_{b-1}(\sqrt{z}) \Gamma(b) \Gamma(2a - b + 1) \\
{}_2F_3\left(\frac{1}{2}, \frac{1}{2}, \frac{3}{2} \middle| z\right) &= \frac{-\log(2\sqrt{z}) + \operatorname{Chi}(2\sqrt{z})}{z} - \frac{\gamma}{z} \\
{}_3F_3\left(\frac{1}{2}, 1, a \middle| 2, 2, a + 1 \middle| z\right) &= \frac{a(-z)^{-a} (\Gamma(a) - \Gamma(a, -z))}{(a-1)^2} + \frac{a(-a+1) (\log(-z) + \operatorname{E}_1(-z) + \gamma)}{z(a^2 - 2a + 1)} - \frac{ae^z}{z(a^2 - 2a + 1)} + \frac{a}{z(a^2 - 2a + 1)}
\end{aligned}$$

5.27.5 References

5.28 Stats

SymPy statistics module

Introduces a random variable type into the SymPy language.

Random variables may be declared using prebuilt functions such as Normal, Exponential, Coin, Die, etc... or built with functions like FiniteRV.

Queries on random expressions can be made using the functions

Expression	Meaning
P(condition)	Probability
E(expression)	Expected value
variance(expression)	Variance
density(expression)	Probability Density Function
sample(expression)	Produce a realization
where(condition)	Where the condition is true

5.28.1 Examples

```
>>> from sympy.stats import P, E, variance, Die, Normal
>>> from sympy import Eq, simplify
>>> X, Y = Die('X', 6), Die('Y', 6) # Define two six sided dice
>>> Z = Normal('Z', 0, 1) # Declare a Normal random variable with mean 0, std 1
>>> P(X>3) # Probability X is greater than 3
1/2
>>> E(X+Y) # Expectation of the sum of two dice
7
>>> variance(X+Y) # Variance of the sum of two dice
35/6
>>> simplify(P(Z>1)) # Probability of Z being greater than 1
-erf(sqrt(2)/2)/2 + 1/2
```

5.28.2 Random Variable Types

Finite Types

`sympy.stats.DiscreteUniform(name, items)`

Create a Finite Random Variable representing a uniform distribution over the input set.

Returns a RandomSymbol.

Examples

```
>>> from sympy.stats import DiscreteUniform, density
>>> from sympy import symbols
```

```
>>> X = DiscreteUniform('X', symbols('a b c')) # equally likely over a, b, c
>>> density(X).dict
{a: 1/3, b: 1/3, c: 1/3}
```

```
>>> Y = DiscreteUniform('Y', list(range(5))) # distribution over a range
>>> density(Y).dict
{0: 1/5, 1: 1/5, 2: 1/5, 3: 1/5, 4: 1/5}
```

`sympy.stats.Die(name, sides=6)`

Create a Finite Random Variable representing a fair die.

Returns a RandomSymbol.

```
>>> from sympy.stats import Die, density
```

```
>>> D6 = Die('D6', 6) # Six sided Die
>>> density(D6).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
```

```
>>> D4 = Die('D4', 4) # Four sided Die
>>> density(D4).dict
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

`sympy.stats.Bernoulli(name, p, succ=1, fail=0)`

Create a Finite Random Variable representing a Bernoulli process.

Returns a RandomSymbol

```
>>> from sympy.stats import Bernoulli, density
>>> from sympy import S
```

```
>>> X = Bernoulli('X', S(3)/4) # 1-0 Bernoulli variable, probability = 3/4
>>> density(X).dict
{0: 1/4, 1: 3/4}
```

```
>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails') # A fair coin toss
>>> density(X).dict
{Heads: 1/2, Tails: 1/2}
```

`sympy.stats.Coin(name, p=1/2)`
 Create a Finite Random Variable representing a Coin toss.
 Probability p is the chance of getting "Heads." Half by default
 Returns a RandomSymbol.

```
>>> from sympy.stats import Coin, density
>>> from sympy import Rational
```

```
>>> C = Coin('C') # A fair coin toss
>>> density(C).dict
{H: 1/2, T: 1/2}
```

```
>>> C2 = Coin('C2', Rational(3, 5)) # An unfair coin
>>> density(C2).dict
{H: 3/5, T: 2/5}
```

`sympy.stats.Binomial(name, n, p, succ=1, fail=0)`
 Create a Finite Random Variable representing a binomial distribution.
 Returns a RandomSymbol.

Examples

```
>>> from sympy.stats import Binomial, density
>>> from sympy import S
```

```
>>> X = Binomial('X', 4, S.Half) # Four "coin flips"
>>> density(X).dict
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}
```

`sympy.stats.Hypergeometric(name, N, m, n)`
 Create a Finite Random Variable representing a hypergeometric distribution.
 Returns a RandomSymbol.

Examples

```
>>> from sympy.stats import Hypergeometric, density
>>> from sympy import S
```

```
>>> X = Hypergeometric('X', 10, 5, 3) # 10 marbles, 5 white (success), 3 draws
>>> density(X).dict
{0: 1/12, 1: 5/12, 2: 5/12, 3: 1/12}
```

`sympy.stats.FiniteRV(name, density)`

Create a Finite Random Variable given a dict representing the density.

Returns a RandomSymbol.

```
>>> from sympy.stats import FiniteRV, P, E
```

```
>>> density = {0: .1, 1: .2, 2: .3, 3: .4}
>>> X = FiniteRV('X', density)
```

```
>>> E(X)
2.0000000000000000
>>> P(X >= 2)
0.7000000000000000
```

Discrete Types

`sympy.stats.Geometric(name, p)`

Create a discrete random variable with a Geometric distribution.

The density of the Geometric distribution is given by

$$f(k) := p(1 - p)^{k-1}$$

Parameters `p`: A probability between 0 and 1

Returns A RandomSymbol.

References

[1] http://en.wikipedia.org/wiki/Geometric_distribution [2] <http://mathworld.wolfram.com/GeometricDistribution.html>

Examples

```
>>> from sympy.stats import Geometric, density, E, variance
>>> from sympy import Symbol, S
```

```
>>> p = S.One / 5
>>> z = Symbol("z")
```

```
>>> X = Geometric("x", p)
```

```
>>> density(X)(z)
(4/5)**(z - 1)/5
```

```
>>> E(X)
5
```

```
>>> variance(X)
20
```

`sympy.stats.Poisson(name, lamda)`

Create a discrete random variable with a Poisson distribution.

The density of the Poisson distribution is given by

$$f(k) := \frac{\lambda^k e^{-\lambda}}{k!}$$

Parameters lamda: Positive number, a rate

Returns A RandomSymbol.

References

[1] http://en.wikipedia.org/wiki/Poisson_distribution [2] <http://mathworld.wolfram.com/PoissonDistribution.html>

Examples

```
>>> from sympy.stats import Poisson, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> rate = Symbol("lambda", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Poisson("x", rate)
```

```
>>> density(X)(z)
lambda**z*exp(-lambda)/factorial(z)
```

```
>>> E(X)
lambda
```

```
>>> simplify(variance(X))
lambda
```

Continuous Types

`sympy.stats.Arcsin(name, a=0, b=1)`

Create a Continuous Random Variable with an arcsin distribution.

The density of the arcsin distribution is given by

$$f(x) := \frac{1}{\pi \sqrt{(x-a)(b-x)}}$$

with $x \in [a, b]$. It must hold that $-\infty < a < b < \infty$.

Parameters **a** : Real number, the left interval boundary

b : Real number, the right interval boundary

Returns A RandomSymbol.

References

[R498] (page 1869)

Examples

```
>>> from sympy.stats import Arcsin, density, cdf
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = Arcsin("x", a, b)
```

```
>>> density(X)(z)
1/(pi*sqrt((-a + z)*(b - z)))
```

```
>>> cdf(X)(z)
Piecewise((0, a > z),
          (2*asin(sqrt((-a + z)/(-a + b)))/pi, b >= z),
          (1, True))
```

`sympy.stats.Benini`(name, alpha, beta, sigma)

Create a Continuous Random Variable with a Benini distribution.

The density of the Benini distribution is given by

$$f(x) := e^{-\alpha \log \frac{x}{\sigma} - \beta \log^2 \left[\frac{x}{\sigma} \right]} \left(\frac{\alpha}{x} + \frac{2\beta \log \frac{x}{\sigma}}{x} \right)$$

This is a heavy-tailed distribution and is also known as the log-Rayleigh distribution.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

sigma : Real number, $\sigma > 0$, a scale

Returns A RandomSymbol.

References

[R499] (page 1869), [R500] (page 1869)

Examples

```
>>> from sympy.stats import Benini, density
>>> from sympy import Symbol, simplify, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Benini("x", alpha, beta, sigma)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/      2*beta*log|-----| - alpha*log|-----| - beta*log 2/ z \
|alpha          \|sigma/          \|sigma/          \|sigma/
|----- + -----|*e
\ z          z          /
```

`sympy.stats.Beta(name, alpha, beta)`

Create a Continuous Random Variable with a Beta distribution.

The density of the Beta distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

with $x \in [0, 1]$.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

Returns A RandomSymbol.

References

[R501] (page 1869), [R502] (page 1869)

Examples

```
>>> from sympy.stats import Beta, density, E, variance
>>> from sympy import Symbol, simplify, pprint, expand_func
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Beta("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1      beta - 1
z      *(-z + 1)
-----
      B(alpha, beta)
```

```
>>> expand_func(simplify(E(X, meijerg=True)))
alpha/(alpha + beta)
```

```
>>> simplify(variance(X, meijerg=True))
alpha*beta/((alpha + beta)**2*(alpha + beta + 1))
```

`sympy.stats.BetaPrime(name, alpha, beta)`

Create a continuous random variable with a Beta prime distribution.

The density of the Beta prime distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1+x)^{-\alpha-\beta}}{B(\alpha, \beta)}$$

with $x > 0$.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

Returns A RandomSymbol.

References

[R503] (page 1869), [R504] (page 1869)

Examples

```
>>> from sympy.stats import BetaPrime, density
>>> from sympy import Symbol, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = BetaPrime("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1      -alpha - beta
z      *(z + 1)
-----
      B(alpha, beta)
```

`sympy.stats.Cauchy(name, x0, gamma)`

Create a continuous random variable with a Cauchy distribution.

The density of the Cauchy distribution is given by

$$f(x) := \frac{1}{\pi} \arctan\left(\frac{x - x_0}{\gamma}\right) + \frac{1}{2}$$

Parameters **x0** : Real number, the location

gamma : Real number, $\gamma > 0$, the scale

Returns A RandomSymbol.

References

[R505] (page 1869), [R506] (page 1869)

Examples

```
>>> from sympy.stats import Cauchy, density
>>> from sympy import Symbol
```

```
>>> x0 = Symbol("x0")
>>> gamma = Symbol("gamma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Cauchy("x", x0, gamma)
```

```
>>> density(X)(z)
1/(pi*gamma*(1 + (-x0 + z)**2/gamma**2))
```

`sympy.stats.Chi(name, k)`

Create a continuous random variable with a Chi distribution.

The density of the Chi distribution is given by

$$f(x) := \frac{2^{1-k/2} x^{k-1} e^{-x^2/2}}{\Gamma(k/2)}$$

with $x \geq 0$.

Parameters **k** : A positive Integer, $k > 0$, the number of degrees of freedom

Returns A RandomSymbol.

References

[R507] (page 1869), [R508] (page 1869)

Examples

```
>>> from sympy.stats import Chi, density, E, std
>>> from sympy import Symbol, simplify
```

```
>>> k = Symbol("k", integer=True)
>>> z = Symbol("z")
```

```
>>> X = Chi("x", k)
```

```
>>> density(X)(z)
2**(-k/2 + 1)*z**(k - 1)*exp(-z**2/2)/gamma(k/2)
```

`sympy.stats.ChiNoncentral`(name, k, l)

Create a continuous random variable with a non-central Chi distribution.

The density of the non-central Chi distribution is given by

$$f(x) := \frac{e^{-(x^2+\lambda^2)/2} x^k \lambda}{(\lambda x)^{k/2}} I_{k/2-1}(\lambda x)$$

with $x \geq 0$. Here, $I_\nu(x)$ is the [modified Bessel function of the first kind](#) (page 513).

Parameters **k** : A positive Integer, $k > 0$, the number of degrees of freedom

l : Shift parameter

Returns A RandomSymbol.

References

[\[R509\]](#) (page 1869)

Examples

```
>>> from sympy.stats import ChiNoncentral, density, E, std
>>> from sympy import Symbol, simplify
```

```
>>> k = Symbol("k", integer=True)
>>> l = Symbol("l")
>>> z = Symbol("z")
```

```
>>> X = ChiNoncentral("x", k, l)
```

```
>>> density(X)(z)
l*z**k*(l*z)**(-k/2)*exp(-l**2/2 - z**2/2)*besseli(k/2 - 1, l*z)
```

`sympy.stats.ChiSquared`(name, k)

Create a continuous random variable with a Chi-squared distribution.

The density of the Chi-squared distribution is given by

$$f(x) := \frac{1}{2^{\frac{k}{2}} \Gamma(\frac{k}{2})} x^{\frac{k}{2}-1} e^{-\frac{x}{2}}$$

with $x \geq 0$.

Parameters **k** : A positive Integer, $k > 0$, the number of degrees of freedom

Returns A RandomSymbol.

References

[R510] (page 1869), [R511] (page 1869)

Examples

```
>>> from sympy.stats import ChiSquared, density, E, variance
>>> from sympy import Symbol, simplify, gammasimp, expand_func
```

```
>>> k = Symbol("k", integer=True, positive=True)
>>> z = Symbol("z")
```

```
>>> X = ChiSquared("x", k)
```

```
>>> density(X)(z)
2**(-k/2)*z**(k/2 - 1)*exp(-z/2)/gamma(k/2)
```

```
>>> gammasimp(E(X))
k
```

```
>>> simplify(expand_func(variance(X)))
2*k
```

`sympy.stats.Dagum(name, p, a, b)`

Create a continuous random variable with a Dagum distribution.

The density of the Dagum distribution is given by

$$f(x) := \frac{ap}{x} \left(\frac{\left(\frac{x}{b}\right)^{ap}}{\left(\left(\frac{x}{b}\right)^a + 1\right)^{p+1}} \right)$$

with $x > 0$.

Parameters **p** : Real number, $p > 0$, a shape

a : Real number, $a > 0$, a shape

b : Real number, $b > 0$, a scale

Returns A RandomSymbol.

References

[R512] (page 1869)

Examples

```
>>> from sympy.stats import Dagum, density, cdf
>>> from sympy import Symbol, simplify
```

```
>>> p = Symbol("p", positive=True)
>>> b = Symbol("b", positive=True)
>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Dagum("x", p, a, b)
```

```
>>> density(X)(z)
a*p*(z/b)**(a*p)*((z/b)**a + 1)**(-p - 1)/z
```

```
>>> cdf(X)(z)
Piecewise(((1 + (z/b)**(-a))**(-p), z >= 0), (0, True))
```

`sympy.stats.Erlang(name, k, l)`

Create a continuous random variable with an Erlang distribution.

The density of the Erlang distribution is given by

$$f(x) := \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}$$

with $x \in [0, \infty]$.

Parameters **k** : Integer

l : Real number, $\lambda > 0$, the rate

Returns A RandomSymbol.

References

[R513] (page 1869), [R514] (page 1869)

Examples

```
>>> from sympy.stats import Erlang, density, cdf, E, variance
>>> from sympy import Symbol, simplify, pprint
```

```
>>> k = Symbol("k", integer=True, positive=True)
>>> l = Symbol("l", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Erlang("x", k, l)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
  k  k - 1  -l*z
l *z      *e
-----
Gamma(k)
```

```
>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
/  -2*I*pi*k
|k*e      *lowergamma(k, l*z)
|-----  for z >= 0
|      Gamma(k + 1)
|
|      0      otherwise
\
```

```
>>> simplify(E(X))
k/l
```

```
>>> simplify(variance(X))
k/l**2
```

`sympy.stats.Exponential(name, rate)`

Create a continuous random variable with an Exponential distribution.

The density of the exponential distribution is given by

$$f(x) := \lambda \exp(-\lambda x)$$

with $x > 0$. Note that the expected value is $1/\lambda$.

Parameters `rate` : A positive Real number, $\lambda > 0$, the rate (or inverse scale/inverse mean)

Returns A RandomSymbol.

References

[R515] (page 1869), [R516] (page 1869)

Examples

```
>>> from sympy.stats import Exponential, density, cdf, E
>>> from sympy.stats import variance, std, skewness
>>> from sympy import Symbol
```

```
>>> l = Symbol("lambda", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Exponential("x", l)
```

```
>>> density(X)(z)
lambda*exp(-lambda*z)
```

```
>>> cdf(X)(z)
Piecewise((1 - exp(-lambda*z), z >= 0), (0, True))
```

```
>>> E(X)
1/lambda
```

```
>>> variance(X)
lambda**(-2)
```

```
>>> skewness(X)
2
```

```
>>> X = Exponential('x', 10)
```

```
>>> density(X)(z)
10*exp(-10*z)
```

```
>>> E(X)
1/10
```

```
>>> std(X)
1/10
```

`sympy.stats.FDistribution(name, d1, d2)`
Create a continuous random variable with a F distribution.

The density of the F distribution is given by

$$f(x) := \frac{\sqrt{\frac{(d_1 x)^{d_1} d_2^{d_2}}{(d_1 x + d_2)^{d_1 + d_2}}}}{x B\left(\frac{d_1}{2}, \frac{d_2}{2}\right)}$$

with $x > 0$.

Parameters **d1** : $d_1 > 0$, where **d_1** is the degrees of freedom (**n_1** - 1)

d2 : $d_2 > 0$, where **d_2** is the degrees of freedom (**n_2** - 1)

Returns A RandomSymbol.

References

[R517] (page 1869), [R518] (page 1869)

Examples

```
>>> from sympy.stats import FDistribution, density
>>> from sympy import Symbol, simplify, pprint
```

```
>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")
```

```
>>> X = FDistribution("x", d1, d2)
```



```

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      d2
      --
      2      /      d1      -d1 - d2
d2  *\ /  (d1*z)  *(d1*z + d2)
-----
              /d1  d2\
            z*B|--, --|
              \2   2 /

```

`sympy.stats.FisherZ(name, d1, d2)`

Create a Continuous Random Variable with an Fisher's Z distribution.

The density of the Fisher's Z distribution is given by

$$f(x) := \frac{2d_1^{d_1/2} d_2^{d_2/2}}{B(d_1/2, d_2/2)} \frac{e^{d_1 z}}{(d_1 e^{2z} + d_2)^{(d_1+d_2)/2}}$$

Parameters **d1** : $d_1 > 0$, degree of freedom

d2 : $d_2 > 0$, degree of freedom

Returns A RandomSymbol.

References

[R519] (page 1869), [R520] (page 1869)

Examples

```

>>> from sympy.stats import FisherZ, density
>>> from sympy import Symbol, simplify, pprint

```

```

>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")

```

```

>>> X = FisherZ("x", d1, d2)

```

```

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      d1      d2
      --      --
      2      2      /      2*z      \      d1*z
2*d1  *d2  *\d1*e  + d2/      *e
-----
              /d1  d2\
            B|--, --|
              \2   2 /

```

`sympy.stats.Frechet(name, a, s=1, m=0)`

Create a continuous random variable with a Frechet distribution.

The density of the Frechet distribution is given by

$$f(x) := \frac{\alpha}{s} \left(\frac{x-m}{s} \right)^{-1-\alpha} e^{-\left(\frac{x-m}{s}\right)^{-\alpha}}$$

with $x \geq m$.

Parameters **a** : Real number, $a \in (0, \infty)$ the shape

s : Real number, $s \in (0, \infty)$ the scale

m : Real number, $m \in (-\infty, \infty)$ the minimum

Returns A RandomSymbol.

References

[R521] (page 1869)

Examples

```
>>> from sympy.stats import Frechet, density, E, std, cdf
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", positive=True)
>>> s = Symbol("s", positive=True)
>>> m = Symbol("m", real=True)
>>> z = Symbol("z")
```

```
>>> X = Frechet("x", a, s, m)
```

```
>>> density(X)(z)
a*((-m + z)/s)**(-a - 1)*exp(-((-m + z)/s)**(-a))/s
```

```
>>> cdf(X)(z)
Piecewise((exp(-((-m + z)/s)**(-a)), m <= z), (0, True))
```

`sympy.stats.Gamma(name, k, theta)`

Create a continuous random variable with a Gamma distribution.

The density of the Gamma distribution is given by

$$f(x) := \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}}$$

with $x \in [0, 1]$.

Parameters **k** : Real number, $k > 0$, a shape

theta : Real number, $\theta > 0$, a scale

Returns A RandomSymbol.

References

[R522] (page 1869), [R523] (page 1869)

Examples

```
>>> from sympy.stats import Gamma, density, cdf, E, variance
>>> from sympy import Symbol, pprint, simplify
```

```
>>> k = Symbol("k", positive=True)
>>> theta = Symbol("theta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Gamma("x", k, theta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
          -z
          ----
      -k  k - 1  theta
theta *z      *e
-----
      Gamma(k)
```

```
>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
/          /      z \
|k*lowergamma|k, ----|
|          \      theta/
<----- for z >= 0
|      Gamma(k + 1)
|
\          0          otherwise
```

```
>>> E(X)
theta*gamma(k + 1)/gamma(k)
```

```
>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
      2
k*theta
```

`sympy.stats.GammaInverse(name, a, b)`

Create a continuous random variable with an inverse Gamma distribution.

The density of the inverse Gamma distribution is given by

$$f(x) := \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp\left(\frac{-\beta}{x}\right)$$

with $x > 0$.

Parameters **a** : Real number, $a > 0$ a shape

b : Real number, $b > 0$ a scale

Returns A RandomSymbol.

References

[R524] (page 1869)

Examples

```
>>> from sympy.stats import GammaInverse, density, cdf, E, variance
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = GammaInverse("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      -b
      ---
      a  -a - 1  z
b *z      *e
-----
Gamma(a)
```

```
>>> cdf(X)(z)
Piecewise((uppergamma(a, b/z)/gamma(a), z > 0), (0, True))
```

`sympy.stats.Kumaraswamy(name, a, b)`

Create a Continuous Random Variable with a Kumaraswamy distribution.

The density of the Kumaraswamy distribution is given by

$$f(x) := abx^{a-1}(1-x^a)^{b-1}$$

with $x \in [0, 1]$.

Parameters **a** : Real number, $a > 0$ a shape

b : Real number, $b > 0$ a shape

Returns A RandomSymbol.

References

[R525] (page 1869)

Examples

```
>>> from sympy.stats import Kumaraswamy, density, E, variance, cdf
>>> from sympy import Symbol, simplify, pprint
```

```
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Kumaraswamy("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
          b - 1
a - 1 / a \
a*b*z    *\ - z + 1/
```

```
>>> cdf(X)(z)
Piecewise((0, z < 0),
          (-(-z**a + 1)**b + 1, z <= 1),
          (1, True))
```

`sympy.stats.Laplace(name, mu, b)`

Create a continuous random variable with a Laplace distribution.

The density of the Laplace distribution is given by

$$f(x) := \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

Parameters **mu** : Real number, the location (mean)

b : Real number, $b > 0$, a scale

Returns A RandomSymbol.

References

[R526] (page 1869), [R527] (page 1870)

Examples

```
>>> from sympy.stats import Laplace, density, cdf
>>> from sympy import Symbol
```

```
>>> mu = Symbol("mu")
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Laplace("x", mu, b)
```

```
>>> density(X)(z)
exp(-Abs(mu - z)/b)/(2*b)
```

```
>>> cdf(X)(z)
Piecewise((exp((-mu + z)/b)/2, mu > z),
          (-exp((mu - z)/b)/2 + 1, True))
```

`sympy.stats.Logistic(name, mu, s)`

Create a continuous random variable with a logistic distribution.

The density of the logistic distribution is given by

$$f(x) := \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2}$$

Parameters **mu** : Real number, the location (mean)

s : Real number, $s > 0$ a scale

Returns A RandomSymbol.

References

[R528] (page 1870), [R529] (page 1870)

Examples

```
>>> from sympy.stats import Logistic, density, cdf
>>> from sympy import Symbol
```

```
>>> mu = Symbol("mu", real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Logistic("x", mu, s)
```

```
>>> density(X)(z)
exp((mu - z)/s)/(s*(exp((mu - z)/s) + 1)**2)
```

```
>>> cdf(X)(z)
1/(exp((mu - z)/s) + 1)
```

`sympy.stats.LogNormal`(name, mean, std)

Create a continuous random variable with a log-normal distribution.

The density of the log-normal distribution is given by

$$f(x) := \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

with $x \geq 0$.

Parameters **mu** : Real number, the log-scale

sigma : Real number, $\sigma^2 > 0$ a shape

Returns A RandomSymbol.

References

[R530] (page 1870), [R531] (page 1870)

Examples

```
>>> from sympy.stats import LogNormal, density
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", real=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = LogNormal("x", mu, sigma)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      2
    -(-mu + log(z))
    -----
      2
    2*sigma
  \ / 2 *e
  -----
    2*\ / pi *sigma*z
```

```
>>> X = LogNormal('x', 0, 1) # Mean 0, standard deviation 1
```

```
>>> density(X)(z)
sqrt(2)*exp(-log(z)**2/2)/(2*sqrt(pi)*z)
```

`sympy.stats.Maxwell(name, a)`

Create a continuous random variable with a Maxwell distribution.

The density of the Maxwell distribution is given by

$$f(x) := \sqrt{\frac{2}{\pi}} \frac{x^2 e^{-x^2/(2a^2)}}{a^3}$$

with $x \geq 0$.

Parameters **a** : Real number, $a > 0$

Returns A RandomSymbol.

References

[R532] (page 1870), [R533] (page 1870)

Examples

```
>>> from sympy.stats import Maxwell, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Maxwell("x", a)
```

```
>>> density(X)(z)
sqrt(2)*z**2*exp(-z**2/(2*a**2))/(sqrt(pi)*a**3)
```

```
>>> E(X)
2*sqrt(2)*a/sqrt(pi)
```

```
>>> simplify(variance(X))
a**2*(-8 + 3*pi)/pi
```

`sympy.stats.Nakagami(name, mu, omega)`

Create a continuous random variable with a Nakagami distribution.

The density of the Nakagami distribution is given by

$$f(x) := \frac{2\mu^\mu}{\Gamma(\mu)\omega^\mu} x^{2\mu-1} \exp\left(-\frac{\mu}{\omega}x^2\right)$$

with $x > 0$.

Parameters **mu** : Real number, $\mu \geq \frac{1}{2}$ a shape

omega : Real number, $\omega > 0$, the spread

Returns A RandomSymbol.

References

[R534] (page 1870)

Examples

```
>>> from sympy.stats import Nakagami, density, E, variance, cdf
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", positive=True)
>>> omega = Symbol("omega", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Nakagami("x", mu, omega)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
          2
      -mu*z
      -----
      mu      -mu  2*mu - 1  omega
2*mu  *omega  *z      *e
-----
      Gamma(mu)
```

```
>>> simplify(E(X, meijerg=True))
sqrt(mu)*sqrt(omega)*gamma(mu + 1/2)/gamma(mu + 1)
```

```
>>> V = simplify(variance(X, meijerg=True))
>>> pprint(V, use_unicode=False)
      2
```

(continues on next page)

(continued from previous page)

$$\text{omega} - \frac{\text{omega} * \text{Gamma}(\text{mu} + 1/2)}{\text{Gamma}(\text{mu}) * \text{Gamma}(\text{mu} + 1)}$$

```
>>> cdf(X)(z)
Piecewise((lowergamma(mu, mu*z**2/omega)/gamma(mu), z > 0),
          (0, True))
```

`sympy.stats.Normal(name, mean, std)`

Create a continuous random variable with a Normal distribution.

The density of the Normal distribution is given by

$$f(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Parameters **mu** : Real number, the mean

sigma : Real number, $\sigma^2 > 0$ the variance

Returns A RandomSymbol.

References

[R535] (page 1870), [R536] (page 1870)

Examples

```
>>> from sympy.stats import Normal, density, E, std, cdf, skewness
>>> from sympy import Symbol, simplify, pprint, factor, together, factor_terms
```

```
>>> mu = Symbol("mu")
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Normal("x", mu, sigma)
```

```
>>> density(X)(z)
sqrt(2)*exp(-(-mu + z)**2/(2*sigma**2))/(2*sqrt(pi)*sigma)
```

```
>>> C = simplify(cdf(X))(z) # it needs a little more help...
>>> pprint(C, use_unicode=False)
      /  _____  \
      | \  2 * (-mu + z) |
erf|-----|
      \      2*sigma      /  1
----- + -
      2                      2
```

```
>>> simplify(skewness(X))
0
```

```
>>> X = Normal("x", 0, 1) # Mean 0, standard deviation 1
>>> density(X)(z)
sqrt(2)*exp(-z**2/2)/(2*sqrt(pi))
```

```
>>> E(2*X + 1)
1
```

```
>>> simplify(std(2*X + 1))
2
```

`sympy.stats.Pareto(name, xm, alpha)`

Create a continuous random variable with the Pareto distribution.

The density of the Pareto distribution is given by

$$f(x) := \frac{\alpha x_m^\alpha}{x^{\alpha+1}}$$

with $x \in [x_m, \infty]$.

Parameters **xm** : Real number, $x_m > 0$, a scale

alpha : Real number, $\alpha > 0$, a shape

Returns A RandomSymbol.

References

[R537] (page 1870), [R538] (page 1870)

Examples

```
>>> from sympy.stats import Pareto, density
>>> from sympy import Symbol
```

```
>>> xm = Symbol("xm", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Pareto("x", xm, beta)
```

```
>>> density(X)(z)
beta*xm**beta*z**(-beta - 1)
```

`sympy.stats.QuadraticU(name, a, b)`

Create a Continuous Random Variable with a U-quadratic distribution.

The density of the U-quadratic distribution is given by

$$f(x) := \alpha(x - \beta)^2$$

with $x \in [a, b]$.

Parameters **a** : Real number

b : Real number, $a < b$

Returns A RandomSymbol.

References

[R539] (page 1870)

Examples

```
>>> from sympy.stats import QuadraticU, density, E, variance
>>> from sympy import Symbol, simplify, factor, pprint
```

```
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")
```

```
>>> X = QuadraticU("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/
| / a b \
|12*|- - - + z|
| \ 2 2 /
<----- for And(b >= z, a <= z)
|
|      3
|      (-a + b)
|
|      0
\      otherwise
```

`sympy.stats.RaisedCosine(name, mu, s)`

Create a Continuous Random Variable with a raised cosine distribution.

The density of the raised cosine distribution is given by

$$f(x) := \frac{1}{2s} \left(1 + \cos \left(\frac{x - \mu}{s} \pi \right) \right)$$

with $x \in [\mu - s, \mu + s]$.

Parameters `mu` : Real number

`s` : Real number, $s > 0$

Returns A RandomSymbol.

References

[R540] (page 1870)

Examples

```
>>> from sympy.stats import RaisedCosine, density, E, variance
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = RaisedCosine("x", mu, s)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/ /pi*(-mu + z)\
|cos|-----| + 1
| \ s /
<----- for And(z >= mu - s, z <= mu + s)
| 2*s
|
\ 0 otherwise
```

`sympy.stats.Rayleigh(name, sigma)`

Create a continuous random variable with a Rayleigh distribution.

The density of the Rayleigh distribution is given by

$$f(x) := \frac{x}{\sigma^2} e^{-x^2/2\sigma^2}$$

with $x > 0$.

Parameters **sigma** : Real number, $\sigma > 0$

Returns A RandomSymbol.

References

[R541] (page 1870), [R542] (page 1870)

Examples

```
>>> from sympy.stats import Rayleigh, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Rayleigh("x", sigma)
```

```
>>> density(X)(z)
z*exp(-z**2/(2*sigma**2))/sigma**2
```

```
>>> E(X)
sqrt(2)*sqrt(pi)*sigma/2
```

```
>>> variance(X)
-pi*sigma**2/2 + 2*sigma**2
```

`sympy.stats.StudentT(name, nu)`

Create a continuous random variable with a student's t distribution.

The density of the student's t distribution is given by

$$f(x) := \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Parameters `nu` : Real number, $\nu > 0$, the degrees of freedom

Returns A `RandomSymbol`.

References

[R543] (page 1870), [R544] (page 1870)

Examples

```
>>> from sympy.stats import StudentT, density, E, variance, cdf
>>> from sympy import Symbol, simplify, pprint
```

```
>>> nu = Symbol("nu", positive=True)
>>> z = Symbol("z")
```

```
>>> X = StudentT("x", nu)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
```

```
      nu      1
      - - - -
      2      2

/      2\
|      z |
| 1 + -- |
\      nu/

-----
\ / nu *B|1/2, --|
      \      2 /
```

```
>>> cdf(X)(z)
1/2 + z*gamma(nu/2 + 1/2)*hyper((1/2, nu/2 + 1/2), (3/2,),
-z**2/nu)/(sqrt(pi)*sqrt(nu)*gamma(nu/2))
```

`sympy.stats.Triangular(name, a, b, c)`

Create a continuous random variable with a triangular distribution.

The density of the triangular distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\ 0 & \text{for } b < x. \end{cases}$$

Parameters **a** : Real number, $a \in (-\infty, \infty)$

b : Real number, $a < b$

c : Real number, $a \leq c \leq b$

Returns A RandomSymbol.

References

[R545] (page 1870), [R546] (page 1870)

Examples

```
>>> from sympy.stats import Triangular, density, E
>>> from sympy import Symbol, pprint
```

```
>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> z = Symbol("z")
```

```
>>> X = Triangular("x", a,b,c)
```

```
>>> pprint(density(X)(z), use_unicode=False)
/      -2*a + 2*z
|----- for And(a <= z, c > z)
| (-a + b)*(-a + c)
|
|      2
|      ----- for c = z
|      -a + b
<
|      2*b - 2*z
|----- for And(b >= z, c < z)
| (-a + b)*(b - c)
|
|      0
\      otherwise
```

`sympy.stats.Uniform(name, left, right)`

Create a continuous random variable with a uniform distribution.

The density of the uniform distribution is given by

$$f(x) := \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

with $x \in [a, b]$.

Parameters **a** : Real number, $-\infty < a$ the left boundary

b : Real number, $a < b < \infty$ the right boundary

Returns A RandomSymbol.

References

[R547] (page 1870), [R548] (page 1870)

Examples

```
>>> from sympy.stats import Uniform, density, cdf, E, variance, skewness
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", negative=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Uniform("x", a, b)
```

```
>>> density(X)(z)
Piecewise((1/(-a + b), (b >= z) & (a <= z)), (0, True))
```

```
>>> cdf(X)(z)
-a/(-a + b) + z/(-a + b)
```

```
>>> simplify(E(X))
a/2 + b/2
```

```
>>> simplify(variance(X))
a**2/12 - a*b/6 + b**2/12
```

`sympy.stats.UniformSum(name, n)`

Create a continuous random variable with an Irwin-Hall distribution.

The probability distribution function depends on a single parameter n which is an integer.

The density of the Irwin-Hall distribution is given by

$$f(x) := \frac{1}{(n-1)!} \sum_{k=0}^{\lfloor x \rfloor} (-1)^k \binom{n}{k} (x-k)^{n-1}$$

Parameters **n** : A positive Integer, $n > 0$

Returns A RandomSymbol.

References

[R549] (page 1870), [R550] (page 1870)

Examples

```
>>> from sympy.stats import UniformSum, density, cdf
>>> from sympy import Symbol, pprint
```

```
>>> n = Symbol("n", integer=True)
>>> z = Symbol("z")
```

```
>>> X = UniformSum("x", n)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
floor(z)
```

$$\sum_{k=0}^{\lfloor z \rfloor} \frac{(-1)^k (-k+z)^{n-1} / n!}{(n-1)!}$$

```
>>> cdf(X)(z)
Piecewise((0, z < 0), (Sum((-1)**_k*(-_k + z)**n*binomial(n, _k),
                           (_k, 0, floor(z)))/factorial(n), n >= z), (1, True))
```

Compute cdf with specific 'x' and 'n' values as follows : `>>> cdf(UniformSum("x", 5), evaluate=False)(2).doit()` 9/40

The argument `evaluate=False` prevents an attempt at evaluation of the sum for general `n`, before the argument 2 is passed.

`sympy.stats.VonMises(name, mu, k)`

Create a Continuous Random Variable with a von Mises distribution.

The density of the von Mises distribution is given by

$$f(x) := \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$$

with $x \in [0, 2\pi]$.

Parameters `mu` : Real number, measure of location

`k` : Real number, measure of concentration

Returns A RandomSymbol.

References

[R551] (page 1870), [R552] (page 1870)

Examples

```
>>> from sympy.stats import VonMises, density, E, variance
>>> from sympy import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu")
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```



```
>>> X = VonMises("x", mu, k)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      k*cos(mu - z)
      e
-----
2*pi*besseli(0, k)
```

`sympy.stats.Weibull(name, alpha, beta)`

Create a continuous random variable with a Weibull distribution.

The density of the Weibull distribution is given by

$$f(x) := \begin{cases} \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Parameters **lambda** : Real number, $\lambda > 0$ a scale

k : Real number, $k > 0$ a shape

Returns A RandomSymbol.

References

[R553] (page 1870), [R554] (page 1870)

Examples

```
>>> from sympy.stats import Weibull, density, E, variance
>>> from sympy import Symbol, simplify
```

```
>>> l = Symbol("lambda", positive=True)
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Weibull("x", l, k)
```

```
>>> density(X)(z)
k*(z/lambda)**(k - 1)*exp(-(z/lambda)**k)/lambda
```

```
>>> simplify(E(X))
lambda*gamma(1 + 1/k)
```

```
>>> simplify(variance(X))
lambda**2*(-gamma(1 + 1/k)**2 + gamma(1 + 2/k))
```

`sympy.stats.WignerSemicircle(name, R)`

Create a continuous random variable with a Wigner semicircle distribution.

The density of the Wigner semicircle distribution is given by

$$f(x) := \frac{2}{\pi R^2} \sqrt{R^2 - x^2}$$

with $x \in [-R, R]$.

Parameters **R** : Real number, $R > 0$, the radius

Returns A *RandomSymbol*.

References

[R555] (page 1870), [R556] (page 1870)

Examples

```
>>> from sympy.stats import WignerSemicircle, density, E
>>> from sympy import Symbol, simplify
```

```
>>> R = Symbol("R", positive=True)
>>> z = Symbol("z")
```

```
>>> X = WignerSemicircle("x", R)
```

```
>>> density(X)(z)
2*sqrt(R**2 - z**2)/(pi*R**2)
```

```
>>> E(X)
0
```

`sympy.stats.ContinuousRV(symbol, density, set=Interval(-oo, oo))`

Create a Continuous Random Variable given the following:

- a symbol
- a probability density function
- set on which the pdf is valid (defaults to entire real line)

Returns a *RandomSymbol*.

Many common continuous random variable types are already implemented. This function should be necessary only very rarely.

Examples

```
>>> from sympy import Symbol, sqrt, exp, pi
>>> from sympy.stats import ContinuousRV, P, E
```

```
>>> x = Symbol("x")
```

```
>>> pdf = sqrt(2)*exp(-x**2/2)/(2*sqrt(pi)) # Normal distribution
>>> X = ContinuousRV(x, pdf)
```

```
>>> E(X)
0
>>> P(X>0)
1/2
```

5.28.3 Interface

`sympy.stats.P(condition, given_condition=None, numsamples=None, evaluate=True, **kwargs)`

Probability that a condition is true, optionally given a second condition

Parameters condition : Combination of Relationals containing RandomSymbols

The condition of which you want to compute the probability

given_condition : Combination of Relationals containing RandomSymbols

A conditional expression. $P(X > 1, X > 0)$ is expectation of $X > 1$ given $X > 0$

numsamples : int

Enables sampling and approximates the probability with this many samples

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral

Examples

```
>>> from sympy.stats import P, Die
>>> from sympy import Eq
>>> X, Y = Die('X', 6), Die('Y', 6)
>>> P(X > 3)
1/2
>>> P(Eq(X, 5), X > 2) # Probability that X == 5 given that X > 2
1/4
>>> P(X > Y)
5/12
```

class `sympy.stats.Probability`

Symbolic expression for the probability.

Examples

```
>>> from sympy.stats import Probability, Normal
>>> from sympy import Integral
>>> X = Normal("X", 0, 1)
>>> prob = Probability(X > 1)
>>> prob
Probability(X > 1)
```

Integral representation:

```
>>> prob.rewrite(Integral)
Integral(sqrt(2)*exp(-_z**2/2)/(2*sqrt(pi)), (_z, 1, oo))
```

Evaluation of the integral:

```
>>> prob.evaluate_integral()
sqrt(2)*(-sqrt(2)*sqrt(pi)*erf(sqrt(2)/2) + sqrt(2)*sqrt(pi))/(4*sqrt(pi))
```

`sympy.stats.E(expr, condition=None, numsamples=None, evaluate=True, **kwargs)`
Returns the expected value of a random expression

Parameters `expr` : Expr containing RandomSymbols

The expression of which you want to compute the expectation value

given : Expr containing RandomSymbols

A conditional expression. $E(X, X > 0)$ is expectation of X given $X > 0$

numsamples : int

Enables sampling and approximates the expectation with this many samples

evalf : Bool (defaults to True)

If sampling return a number rather than a complex expression

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral

Examples

```
>>> from sympy.stats import E, Die
>>> X = Die('X', 6)
>>> E(X)
7/2
>>> E(2*X + 1)
8
```

```
>>> E(X, X > 3) # Expectation of X given that it is above 3
5
```

class `sympy.stats.Expectation`
Symbolic expression for the expectation.

Examples

```
>>> from sympy.stats import Expectation, Normal, Probability
>>> from sympy import symbols, Integral
>>> mu = symbols("mu")
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)
>>> Expectation(X)
Expectation(X)
>>> Expectation(X).evaluate_integral().simplify()
mu
```

To get the integral expression of the expectation:

```
>>> Expectation(X).rewrite(Integral)
Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/(2*sqrt(pi)*sigma), (X, -oo,
↪ oo))
```

The same integral expression, in more abstract terms:

```
>>> Expectation(X).rewrite(Probability)
Integral(x*Probability(Eq(X, x)), (x, -oo, oo))
```

This class is aware of some properties of the expectation:

```
>>> from sympy.abc import a
>>> Expectation(a*X)
Expectation(a*X)
>>> Y = Normal("Y", 0, 1)
>>> Expectation(X + Y)
Expectation(X + Y)
```

To expand the Expectation into its expression, use `doit()`:

```
>>> Expectation(X + Y).doit()
Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y).doit()
a*Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y)
Expectation(a*X + Y)
```

`sympy.stats.density(expr, condition=None, evaluate=True, numsamples=None, **kwargs)`

Probability density of a random expression, optionally given a second condition.

This density will take on different forms for different types of probability spaces. Discrete variables produce Dicts. Continuous variables produce Lambdas.

Parameters `expr` : Expr containing RandomSymbols

The expression of which you want to compute the density value

condition : Relational containing RandomSymbols

A conditional expression. `density(X > 1, X > 0)` is density of $X > 1$ given $X > 0$

numsamples : int

Enables sampling and approximates the density with this many samples

Examples

```
>>> from sympy.stats import density, Die, Normal
>>> from sympy import Symbol
```

```
>>> x = Symbol('x')
>>> D = Die('D', 6)
>>> X = Normal(x, 0, 1)
```

```
>>> density(D).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
>>> density(2*D).dict
{2: 1/6, 4: 1/6, 6: 1/6, 8: 1/6, 10: 1/6, 12: 1/6}
>>> density(X)(x)
sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

`sympy.stats.given(expr, condition=None, **kwargs)`

Conditional Random Expression From a random expression and a condition on that expression creates a new probability space from the condition and returns the same expression on that conditional probability space.

Examples

```
>>> from sympy.stats import given, density, Die
>>> X = Die('X', 6)
>>> Y = given(X, X > 3)
>>> density(Y).dict
{4: 1/3, 5: 1/3, 6: 1/3}
```

Following convention, if the condition is a random symbol then that symbol is considered fixed.

```
>>> from sympy.stats import Normal
>>> from sympy import pprint
>>> from sympy.abc import z
```

```
>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
>>> pprint(density(X + Y, Y)(z), use_unicode=False)
      2
    -(-Y + z)
    -----
      2
    \ 2 *e
    -----
      2*\  pi
```

`sympy.stats.where(condition, given_condition=None, **kwargs)`

Returns the domain where a condition is True.

Examples

```
>>> from sympy.stats import where, Die, Normal
>>> from sympy import symbols, And
```

```
>>> D1, D2 = Die('a', 6), Die('b', 6)
>>> a, b = D1.symbol, D2.symbol
>>> X = Normal('x', 0, 1)
```

```
>>> where(X**2<1)
Domain: (-1 < x) & (x < 1)
```

```
>>> where(X**2<1).set
Interval.open(-1, 1)
```

```
>>> where(And(D1<=D2 , D2<3))
Domain: (Eq(a, 1) & Eq(b, 1)) | (Eq(a, 1) & Eq(b, 2)) | (Eq(a, 2) & Eq(b, 2))
```

`sympy.stats.variance(X, condition=None, **kwargs)`
 Variance of a random expression
 Expectation of $(X-E(X))^2$

Examples

```
>>> from sympy.stats import Die, E, Bernoulli, variance
>>> from sympy import simplify, Symbol
```

```
>>> X = Die('X', 6)
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> variance(2*X)
35/3
```

```
>>> simplify(variance(B))
p*(-p + 1)
```

class `sympy.stats.Variance`
 Symbolic expression for the variance.

Examples

```
>>> from sympy import symbols, Integral
>>> from sympy.stats import Normal, Expectation, Variance, Probability
>>> mu = symbols("mu", positive=True)
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)
>>> Variance(X)
Variance(X)
>>> Variance(X).evaluate_integral()
sigma**2
```

Integral representation of the underlying calculations:

```
>>> Variance(X).rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/
↳ (2*sqrt(pi)*sigma), (X, -oo, oo)))**2*exp(-(X - mu)**2/(2*sigma**2))/
↳ (2*sqrt(pi)*sigma), (X, -oo, oo))
```

Integral representation, without expanding the PDF:

```
>>> Variance(X).rewrite(Probability)
-Integral(x*Probability(Eq(X, x)), (x, -oo, oo))**2 +
↳ Integral(x**2*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the variance in terms of the expectation

```
>>> Variance(X).rewrite(Expectation)
-Expectation(X)**2 + Expectation(X**2)
```

Some transformations based on the properties of the variance may happen:

```
>>> from sympy.abc import a
>>> Y = Normal("Y", 0, 1)
>>> Variance(a*X)
Variance(a*X)
```

To expand the variance in its expression, use `doit()`:

```
>>> Variance(a*X).doit()
a**2*Variance(X)
>>> Variance(X + Y)
Variance(X + Y)
>>> Variance(X + Y).doit()
2*Covariance(X, Y) + Variance(X) + Variance(Y)
```

`sympy.stats.covariance(X, Y, condition=None, **kwargs)`

Covariance of two random expressions

The expectation that the two variables will rise and fall together

$\text{Covariance}(X, Y) = E((X - E(X)) * (Y - E(Y)))$

Examples

```
>>> from sympy.stats import Exponential, covariance
>>> from sympy import Symbol
```

```
>>> rate = Symbol('lambda', positive=True, real=True, finite=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)
```

```
>>> covariance(X, X)
lambda**(-2)
>>> covariance(X, Y)
0
>>> covariance(X, Y + rate*X)
1/lambda
```

class `sympy.stats.Covariance`

Symbolic expression for the covariance.

Examples

```
>>> from sympy.stats import Covariance
>>> from sympy.stats import Normal
>>> X = Normal("X", 3, 2)
>>> Y = Normal("Y", 0, 1)
>>> Z = Normal("Z", 0, 1)
>>> W = Normal("W", 0, 1)
>>> cexpr = Covariance(X, Y)
>>> cexpr
Covariance(X, Y)
```

Evaluate the covariance, X and Y are independent, therefore zero is the result:


```
>>> cexpr.evaluate_integral()
0
```

Rewrite the covariance expression in terms of expectations:

```
>>> from sympy.stats import Expectation
>>> cexpr.rewrite(Expectation)
Expectation(X*Y) - Expectation(X)*Expectation(Y)
```

In order to expand the argument, use `doit()`:

```
>>> from sympy.abc import a, b, c, d
>>> Covariance(a*X + b*Y, c*Z + d*W)
Covariance(a*X + b*Y, c*Z + d*W)
>>> Covariance(a*X + b*Y, c*Z + d*W).doit()
a*c*Covariance(X, Z) + a*d*Covariance(W, X) + b*c*Covariance(Y, Z) +
↪ b*d*Covariance(W, Y)
```

This class is aware of some properties of the covariance:

```
>>> Covariance(X, X).doit()
Variance(X)
>>> Covariance(a*X, b*Y).doit()
a*b*Covariance(X, Y)
```

`sympy.stats.std(X, condition=None, **kwargs)`
 Standard Deviation of a random expression
 Square root of the Expectation of $(X-E(X))^2$

Examples

```
>>> from sympy.stats import Bernoulli, std
>>> from sympy import Symbol, simplify
```

```
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> simplify(std(B))
sqrt(p*(-p + 1))
```

`sympy.stats.sample(expr, condition=None, **kwargs)`
 A realization of the random expression

Examples

```
>>> from sympy.stats import Die, sample
>>> X, Y, Z = Die('X', 6), Die('Y', 6), Die('Z', 6)
```

```
>>> die_roll = sample(X + Y + Z) # A random realization of three dice
```

`sympy.stats.sample_iter(expr, condition=None, numsamples=oo, **kwargs)`
 Returns an iterator of realizations from the expression given a condition

expr: Random expression to be realized condition: A conditional expression (optional)
numsamples: Length of the iterator (defaults to infinity)

See also:

Sample, sampling_P, sampling_E, sample_iter_lambdify, sample_iter_subs

Examples

```
>>> from sympy.stats import Normal, sample_iter
>>> X = Normal('X', 0, 1)
>>> expr = X*X + 3
>>> iterator = sample_iter(expr, numsamples=3)
>>> list(iterator)
[12, 4, 7]
```

5.28.4 Mechanics

SymPy Stats employs a relatively complex class hierarchy.

RandomDomains are a mapping of variables to possible values. For example we might say that the symbol `Symbol('x')` can take on the values $\{1, 2, 3, 4, 5, 6\}$.

class sympy.stats.rv.RandomDomain

A PSpace, or Probability Space, combines a RandomDomain with a density to provide probabilistic information. For example the above domain could be enhanced by a finite density $\{1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6\}$ to fully define the roll of a fair die named `x`.

class sympy.stats.rv.PSpace

A RandomSymbol represents the PSpace's symbol 'x' inside of SymPy expressions.

class sympy.stats.rv.RandomSymbol

The RandomDomain and PSpace classes are almost never directly instantiated. Instead they are subclassed for a variety of situations.

RandomDomains and PSpaces must be sufficiently general to represent domains and spaces of several variables with arbitrarily complex densities. This generality is often unnecessary. Instead we often build SingleDomains and SinglePSpaces to represent single, univariate events and processes such as a single die or a single normal variable.

class sympy.stats.rv.SinglePSpace**class sympy.stats.rv.SingleDomain**

Another common case is to collect together a set of such univariate random variables. A collection of independent SinglePSpaces or SingleDomains can be brought together to form a ProductDomain or ProductPSpace. These objects would be useful in representing three dice rolled together for example.

class sympy.stats.rv.ProductDomain**class sympy.stats.rv.ProductPSpace**

The Conditional adjective is added whenever we add a global condition to a RandomDomain or PSpace. A common example would be three independent dice where we know their sum to be greater than 12.

class sympy.stats.rv.**ConditionalDomain**

We specialize further into Finite and Continuous versions of these classes to represent finite (such as dice) and continuous (such as normals) random variables.

class sympy.stats.frv.**FiniteDomain**

class sympy.stats.frv.**FinitePSpace**

class sympy.stats.crv.**ContinuousDomain**

class sympy.stats.crv.**ContinuousPSpace**

Additionally there are a few specialized classes that implement certain common random variable types. There is for example a `DiePSpace` that implements `SingleFinitePSpace` and a `NormalPSpace` that implements `SingleContinuousPSpace`.

class sympy.stats.frv_types.**DiePSpace**

class sympy.stats.crv_types.**NormalPSpace**

RandomVariables can be extracted from these objects using the `PSpace.values` method.

As previously mentioned SymPy Stats employs a relatively complex class structure. Inheritance is widely used in the implementation of end-level classes. This tactic was chosen to balance between the need to allow SymPy to represent arbitrarily defined random variables and optimizing for common cases. This complicates the code but is structured to only be important to those working on extending SymPy Stats to other random variable types.

Users will not use this class structure. Instead these mechanics are exposed through variable creation functions `Die`, `Coin`, `FiniteRV`, `Normal`, `Exponential`, etc.... These build the appropriate `SinglePSpaces` and return the corresponding `RandomVariable`. Conditional and Product spaces are formed in the natural construction of SymPy expressions and the use of interface functions `E`, `Given`, `Density`, etc....

`sympy.stats.Die()`

`sympy.stats.Normal()`

There are some additional functions that may be useful. They are largely used internally.

`sympy.stats.rv.random_symbols(expr)`

Returns all `RandomSymbols` within a SymPy Expression.

`sympy.stats.rv.pspace(expr)`

Returns the underlying Probability Space of a random expression.

For internal use.

Examples

```
>>> from sympy.stats import pspace, Normal
>>> from sympy.stats.rv import ProductPSpace
>>> X = Normal('X', 0, 1)
>>> pspace(2*X + 1) == X.pspace
True
```

`sympy.stats.rv.rs_swap(a, b)`

Build a dictionary to swap `RandomSymbols` based on their underlying symbol.

i.e. if `X = ('x', pspace1)` and `Y = ('x', pspace2)` then `X` and `Y` match and the key, value pair `{X:Y}` will appear in the result

Inputs: collections a and b of random variables which share common symbols
Output: dict mapping RVs in a to RVs in b

5.29 ODE

5.29.1 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions (unlike [Hint Functions](#) (page 1238), below) are intended for use by ordinary users of SymPy.

`dsolve()`

`sympy.solvers.ode.dsolve(eq, func=None, hint='default', simplify=True, ics=None, xi=None, eta=None, x0=0, n=6, **kwargs)`

Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.

Examples

```
>>> from sympy import Function, dsolve, Eq, Derivative, sin, cos, symbols
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))
```

```
>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='1st_exact')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> dsolve(eq, hint='almost_linear')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> t = symbols('t')
>>> x, y = symbols('x, y', function=True)
>>> eq = (Eq(Derivative(x(t), t), 12*t*x(t) + 8*y(t)), Eq(Derivative(y(t), t),
↪ 21*x(t) + 7*t*y(t)))
>>> dsolve(eq)
[Eq(x(t), C1*x0 + C2*x0*Integral(8*exp(Integral(7*t, t))*exp(Integral(12*t, t))/
↪ x0**2, t)),
Eq(y(t), C1*y0 + C2(y0*Integral(8*exp(Integral(7*t, t))*exp(Integral(12*t, t))/
↪ x0**2, t) +
exp(Integral(7*t, t))*exp(Integral(12*t, t))/x0))]
>>> eq = (Eq(Derivative(x(t), t), x(t)*y(t)*sin(t)), Eq(Derivative(y(t), t),
↪ y(t)**2*sin(t)))
>>> dsolve(eq)
{Eq(x(t), -exp(C1)/(C2*exp(C1) - cos(t))), Eq(y(t), -1/(C1 - cos(t)))}
```

For Single Ordinary Differential Equation

It is classified under this when number of equation in eq is one. **Usage**

`dsolve(eq, f(x), hint)` -> Solve ordinary differential equation `eq` for function `f(x)`, using method `hint`.

Details

eq can be any supported ordinary differential equation (see the [ode](#) (page 1283) docstring for supported methods). This can either be an [Equality](#) (page 162), or an expression, which is assumed to be equal to 0.

f(x) is a function of one variable whose derivatives in that variable make up the ordinary differential equation `eq`. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it couldn't be detected).

hint is the solving method that you want dsolve to use. Use `classify_ode(eq, f(x))` to get all of the possible hints for an ODE. The default hint, `default`, will use whatever hint is returned first by `classify_ode()` (page 1233). See Hints below for more options that you can use for hint.

simplify enables simplification by `odesimp()` (page 1238). See its docstring for more information. Turn this off, for example, to disable solving of solutions for `func` or simplification of arbitrary constants. It will still integrate with this hint. Note that the solution may contain more arbitrary constants than the order of the ODE with this option enabled.

xi and eta are the infinitesimal functions of an ordinary differential equation. They are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. The user can specify values for the infinitesimals. If nothing is specified, `xi` and `eta` are calculated using `infinitesimals()` (page 1236) with the help of various heuristics.

ics is the set of initial/boundary conditions for the differential equation. It should be given in the form of `{f(x0): x1, f(x).diff(x).subs(x, x2): x3}` and so on. For power series solutions, if no initial conditions are specified `f(0)` is assumed to be `C0` and the power series solution is calculated about 0.

x0 is the point about which the power series solution of a differential equation is to be evaluated.

n gives the exponent of the dependent variable up to which the power series solution of a differential equation is to be evaluated.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to `dsolve()` (page 1230):

default: This uses whatever hint is returned first by `classify_ode()` (page 1233). This is the default argument to `dsolve()` (page 1230).

all: To make `dsolve()` (page 1230) apply all relevant classification hints, use `dsolve(ODE, func, hint="all")`. This will return a dictionary of `hint:solution` terms. If a hint causes `dsolve` to raise the `NotImplementedError`, value of that hint's key will be the exception object raised. The dictionary will also include some special keys:

- `order`: The order of the ODE. See also `ode_order()` (page 1310) in `deutils.py`.
- `best`: The simplest hint; what would be returned by best below.

- **best_hint**: The hint that would produce the solution given by **best**. If more than one hint produces the best solution, the first one in the tuple returned by `classify_ode()` (page 1233) is chosen.
- **default**: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by `classify_ode()` (page 1233).

all_Integral: This is the same as **all**, except if a hint also has a corresponding `_Integral` hint, it only returns the `_Integral` hint. This is useful if **all** causes `dsolve()` (page 1230) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than **all**, because `integrate()` (page 124) is an expensive routine.

best: To have `dsolve()` (page 1230) try all methods and return the simplest one. This takes into account whether the solution is solvable in the function, whether it contains any Integral classes (i.e. unevaluable integrals), and which one is the shortest in size.

See also the `classify_ode()` (page 1233) docstring for more info on hints, and the `ode` (page 1283) docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x # x is the independent variable
>>> f = Function("f")(x) # f is a function of x
>>> # f_ will be the derivative of f with respect to x
>>> f_ = Derivative(f, x)
```

- See `test_ode.py` for many tests, which serves also as a set of examples for how to use `dsolve()` (page 1230).
- `dsolve()` (page 1230) always returns an `Equality` (page 162) class (except for the case when the hint is **all** or **all_Integral**). If possible, it solves the solution explicitly for the function being solved for. Otherwise, it returns an implicit solution.
- Arbitrary constants are symbols named `C1`, `C2`, and so on.
- Because all solutions should be mathematically equivalent, some hints may return the exact same result for an ODE. Often, though, two different hints will return the same solution formatted differently. The two should be equivalent. Also note that sometimes the values of the arbitrary constants in two different solutions may not be the same, because one constant may have “absorbed” other constants into it.
- Do `help(ode.ode_<hintname>)` to get help more information on a specific hint, where `<hintname>` is the name of a hint without `_Integral`.

For System Of Ordinary Differential Equations

Usage

`dsolve(eq, func)` -> Solve a system of ordinary differential equations `eq` for `func` being list of functions including $x(t)$, $y(t)$, $z(t)$ where number of functions in the list depends upon the number of equations provided in `eq`.

Details

eq can be any supported system of ordinary differential equations This can either be an [Equality](#) (page 162), or an expression, which is assumed to be equal to 0.

func holds $x(t)$ and $y(t)$ being functions of one variable which together with some of their derivatives make up the system of ordinary differential equation eq. It is not necessary to provide this; it will be autodetected (and an error raised if it couldn't be detected).

Hints

The hints are formed by parameters returned by `classify_sysode`, combining them give hints name used later for forming method name.

`classify_ode()`

`sympy.solvers.ode.classify_ode(eq, func=None, dict=False, ics=None, **kwargs)`

Returns a tuple of possible [dsolve\(\)](#) (page 1230) classifications for an ODE.

The tuple is ordered so that first item is the classification that [dsolve\(\)](#) (page 1230) uses to solve the ODE by default. In general, classifications at the near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make [dsolve\(\)](#) (page 1230) use a different classification, use `dsolve(ODE, func, hint=<classification>)`. See also the [dsolve\(\)](#) (page 1230) docstring for different meta-hints you can use.

If `dict` is true, [classify_ode\(\)](#) (page 1233) will return a dictionary of `hint:match` expression terms. This is intended for internal use by [dsolve\(\)](#) (page 1230). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hint-name` is the name of the hint without `_Integral`.

See [allhints](#) (page 1238) or the [ode](#) (page 1283) docstring for a list of all supported hints that can be returned from [classify_ode\(\)](#) (page 1233).

Notes

These are remarks on hint names.

`_Integral`

If a classification has `_Integral` at the end, it will return the expression with an unevaluated [Integral](#) (page 690) class in it. Note that a hint may do this anyway if [integrate\(\)](#) (page 124) cannot do the integral, though just using an `_Integral` will do so much faster. Indeed, an `_Integral` hint will always be faster than its corresponding hint without `_Integral` because [integrate\(\)](#) (page 124) is an expensive routine. If [dsolve\(\)](#) (page 1230) hangs, it is probably because [integrate\(\)](#) (page 124) is hanging on a tough or impossible integral. Try using an `_Integral` hint or `all_Integral` to get it return something.

Note that some hints do not have `_Integral` counterparts. This is because [integrate\(\)](#) is not used in solving the ODE for those method. For example, n th order linear homogeneous ODEs with constant coefficients do not require integration to solve, so there is no

`nth_linear_homogeneous_constant_coeff_Integrate` hint. You can easily evaluate any unevaluated `Integral` (page 690)s in an expression by doing `expr.doit()`.

Ordinals

Some hints contain an ordinal such as `1st_linear`. This is to help differentiate them from other hints, as well as from other methods that may not be implemented yet. If a hint has `nth` in it, such as the `nth_linear` hints, this means that the method used to applies to ODEs of any order.

indep and dep

Some hints contain the words `indep` or `dep`. These reference the independent variable and the dependent function, respectively. For example, if an ODE is in terms of $f(x)$, then `indep` will refer to x and `dep` will refer to f .

subs

If a hints has the word `subs` in it, it means the the ODE is solved by substituting the expression given after the word `subs` for a single dummy variable. This is usually in terms of `indep` and `dep` as above. The substituted expression will be written only in characters allowed for names of Python objects, meaning operators will be spelled out. For example, `indep/dep` will be written as `indep_div_dep`.

coeff

The word `coeff` in a hint refers to the coefficients of something in the ODE, usually of the derivative terms. See the docstring for the individual methods for more info (`help(ode)`). This is contrast to `coefficients`, as in `undetermined_coefficients`, which refers to the common name of a method.

_best

Methods that have more than one fundamental way to solve will have a hint for each sub-method and a `_best` meta-classification. This will evaluate all hints and return the best, using the same considerations as the normal best meta-hint.

Examples

```
>>> from sympy import Function, classify_ode, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('separable', '1st_linear', '1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep',
'1st_power_series', 'lie_group',
'nth_linear_constant_coeff_homogeneous',
'separable_Integral', '1st_linear_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_constant_coeff_variation_of_parameters_Integral')
```


checkodesol()

`sympy.solvers.ode.checkodesol(ode, sol, func=None, order='auto', solve_for_func=True)`

Substitutes `sol` into `ode` and checks that the result is 0.

This only works when `func` is one function, like $f(x)$. `sol` can be a single solution or a list of solutions. Each solution may be an [Equality](#) (page 162) that the solution satisfies, e.g. `Eq(f(x), C1)`, `Eq(f(x) + C1, 0)`; or simply an [Expr](#) (page 111), e.g. `f(x) - C1`. In most cases it will not be necessary to explicitly identify the function, but if the function cannot be inferred from the original equation it can be supplied through the `func` argument.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

It tries the following methods, in order, until it finds zero equivalence:

1. Substitute the solution for f in the original equation. This only works if `ode` is solved for f . It will attempt to solve it first unless `solve_for_func == False`.
2. Take n derivatives of the solution, where n is the order of `ode`, and check to see if that is equal to the solution. This only works on exact ODEs.
3. Take the 1st, 2nd, ..., n th derivatives of the solution, each time solving for the derivative of f of that order (this will always be possible because f is a linear operator). Then back substitute each derivative into `ode` in reverse order.

This function returns a tuple. The first item in the tuple is `True` if the substitution results in 0, and `False` otherwise. The second item in the tuple is what the substitution results in. It should always be 0 if the first item is `True`. Sometimes this function will return `False` even when an expression is identically equal to 0. This happens when [simplify\(\)](#) (page 1154) does not reduce the expression to 0. If an expression returned by this function vanishes identically, then `sol` really is a solution to the ode.

If this function seems to hang, it is probably because of a hard simplification.

To use this function to test, test the first item of the tuple.

Examples

```
>>> from sympy import Eq, Function, checkodesol, symbols
>>> x, C1 = symbols('x,C1')
>>> f = Function('f')
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> assert checkodesol(f(x).diff(x), C1)[0]
>>> assert not checkodesol(f(x).diff(x), x)[0]
>>> checkodesol(f(x).diff(x, 2), x**2)
(False, 2)
```

homogeneous_order()

`sympy.solvers.ode.homogeneous_order(eq, *symbols)`

Returns the order n if g is homogeneous and `None` if it is not homogeneous.

Determines if a function is homogeneous and if so of what order. A function $f(x, y, \dots)$ is homogeneous of order n if $f(tx, ty, \dots) = t^n f(x, y, \dots)$.

If the function is of two variables, $F(x, y)$, then f being homogeneous of any order is equivalent to being able to rewrite $F(x, y)$ as $G(x/y)$ or $H(y/x)$. This fact is used to solve 1st order ordinary differential equations whose coefficients are homogeneous of the same order (see the docstrings of `ode_1st_homogeneous_coeff_subs_dep_div_indep()` and `ode_1st_homogeneous_coeff_subs_indep_div_dep()`).

Symbols can be functions, but every argument of the function must be a symbol, and the arguments of the function that appear in the expression must match those given in the list of symbols. If a declared function appears with different arguments than given in the list of symbols, `None` is returned.

Examples

```
>>> from sympy import Function, homogeneous_order, sqrt
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> homogeneous_order(f(x), f(x)) is None
True
>>> homogeneous_order(f(x,y), f(y, x), x, y) is None
True
>>> homogeneous_order(f(x), f(x), x)
1
>>> homogeneous_order(x**2*f(x)/sqrt(x**2+f(x)**2), x, f(x))
2
>>> homogeneous_order(x**2+f(x), x, f(x)) is None
True
```

infinitesimals()

`sympy.solvers.ode.infinitesimals(eq, func=None, order=None, hint='default', match=None)`

The infinitesimal functions of an ordinary differential equation, $\xi(x, y)$ and $\eta(x, y)$, are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. So, the ODE $y' = f(x, y)$ would admit a Lie group $x^* = X(x, y; \varepsilon) = x + \varepsilon \xi(x, y)$, $y^* = Y(x, y; \varepsilon) = y + \varepsilon \eta(x, y)$ such that $(y^*)' = f(x^*, y^*)$. A change of coordinates, to $r(x, y)$ and $s(x, y)$, can be performed so this Lie group becomes the translation group, $r^* = r$ and $s^* = s + \varepsilon$. They are tangents to the coordinate curves of the new system.

Consider the transformation $(x, y) \rightarrow (X, Y)$ such that the differential equation remains invariant. ξ and η are the tangents to the transformed coordinates X and Y , at $\varepsilon = 0$.

$$\left(\frac{\partial X(x, y; \varepsilon)}{\partial \varepsilon} \right) \Big|_{\varepsilon=0} = \xi, \quad \left(\frac{\partial Y(x, y; \varepsilon)}{\partial \varepsilon} \right) \Big|_{\varepsilon=0} = \eta,$$

The infinitesimals can be found by solving the following PDE:

```
>>> from sympy import Function, diff, Eq, pprint
>>> from sympy.abc import x, y
>>> xi, eta, h = map(Function, ['xi', 'eta', 'h'])
>>> h = h(x, y) # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq(eta.diff(x) + (eta.diff(y) - xi.diff(x))*h
... - (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
```

(continues on next page)

(continued from previous page)

```
>>> pprint(genform)
/d      d      \
|--(eta(x, y)) - --(xi(x, y))|*h(x, y) - eta(x, y)*--(h(x, y)) - h2(x, y)*--(x
\dy      dx      /      dy
i(x, y)) - xi(x, y)*--(h(x, y)) + --(eta(x, y)) = 0
      dx      dx
```

Solving the above mentioned PDE is not trivial, and can be solved only by making intelligent assumptions for ξ and η (heuristics). Once an infinitesimal is found, the attempt to find more heuristics stops. This is done to optimise the speed of solving the differential equation. If a list of all the infinitesimals is needed, `hint` should be flagged as `all`, which gives the complete list of infinitesimals. If the infinitesimals for a particular heuristic needs to be found, it can be passed as a flag to `hint`.

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

Examples

```
>>> from sympy import Function, diff
>>> from sympy.solvers.ode import infinitesimals
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq)
[{eta(x, f(x)): exp(x**3/3), xi(x, f(x)): 0}]
```

checkinfsol()

`sympy.solvers.ode.checkinfsol(eq, infinitesimals, func=None, order=None)`

This function is used to check if the given infinitesimals are the actual infinitesimals of the given first order differential equation. This method is specific to the Lie Group Solver of ODEs.

As of now, it simply checks, by substituting the infinitesimals in the partial differential equation.

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi \frac{\partial h}{\partial x} - \eta \frac{\partial h}{\partial y} = 0$$

where η , and ξ are the infinitesimals and $h(x, y) = \frac{dy}{dx}$

The infinitesimals should be given in the form of a list of dicts `[{xi(x, y): inf, eta(x, y): inf}]`, corresponding to the output of the function `infinitesimals`. It returns a list of values of the form `[(True/False, sol)]` where `sol` is the value obtained after substituting the infinitesimals in the PDE. If it is `True`, then `sol` would be 0.

5.29.2 Hint Functions

These functions are intended for internal use by `dsolve()` (page 1230) and others. Unlike [User Functions](#) (page 1230), above, these are not intended for every-day use by ordinary SymPy users. Instead, functions such as `dsolve()` (page 1230) should be used. Nonetheless, these functions contain useful information in their docstrings on the various ODE solving methods. For this reason, they are documented here.

`allhints`

`sympy.solvers.ode.allhints = ('separable', '1st_exact', '1st_linear', 'Bernoulli', 'Riccat`

This is a list of hints in the order that they should be preferred by `classify_ode()` (page 1233). In general, hints earlier in the list should produce simpler solutions than those later in the list (for ODEs that fit both). For now, the order of this list is based on empirical observations by the developers of SymPy.

The hint used by `dsolve()` (page 1230) for a specific ODE can be overridden (see the docstring).

In general, `_Integral` hints are grouped at the end of the list, unless there is a method that returns an unevaluable integral most of the time (which go near the end of the list anyway). `default`, `all`, `best`, and `all_Integral` meta-hints should not be included in this list, but `_best` and `_Integral` hints should be included.

`odesimp`

`sympy.solvers.ode.odesimp(eq, func, order, constants, hint)`

Simplifies ODEs, including trying to solve for `func` and running `constantsimp()` (page 1240).

It may use knowledge of the type of solution that the hint returns to apply additional simplifications.

It also attempts to integrate any [Integral](#) (page 690)s in the expression, if the hint is not an `_Integral` hint.

This function should have no effect on expressions returned by `dsolve()` (page 1230), as `dsolve()` (page 1230) already calls `odesimp()` (page 1238), but the individual hint functions do not call `odesimp()` (page 1238) (because the `dsolve()` (page 1230) wrapper does). Therefore, this function is designed for mainly internal use.

Examples

```
>>> from sympy import sin, symbols, dsolve, pprint, Function
>>> from sympy.solvers.ode import odesimp
>>> x, u2, C1 = symbols('x,u2,C1')
>>> f = Function('f')
```

```
>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral',
... simplify=False)
>>> pprint(eq, wrap_line=False)
      x
```

(continues on next page)

(continued from previous page)

$$\log(f(x)) = \log(C1) + \frac{\int \frac{1}{\sqrt{-u^2 + \frac{1}{\sin^2(u)}}} du}{u^2}$$

```
>>> pprint(odesimp(eq, f(x), 1, {C1},
... hint='lst_homogeneous_coeff_subs_indep_div_dep'
... ))
x
----- = C1
/f(x)\
tan|----|
\2*x /
```

constant_renumber

`sympy.solvers.ode.constant_renumber(expr, symbolname, startnumber, endnumber)`

Renumber arbitrary constants in `expr` to have numbers 1 through N where N is `endnumber - startnumber + 1` at most. In the process, this reorders expression terms in a standard way.

This is a simple function that goes through and rennumbers any [Symbol](#) (page 133) with a name in the form `symbolname + num` where `num` is in the range from `startnumber` to `endnumber`.

Symbols are renumbered based on `.sort_key()`, so they should be numbered roughly in the order that they appear in the final, printed expression. Note that this ordering is based in part on hashes, so it can produce different results on different machines.

The structure of this function is very similar to that of `constantsimp()` (page 1240).

Examples

```
>>> from sympy import symbols, Eq, pprint
>>> from sympy.solvers.ode import constant_renumber
>>> x, C0, C1, C2, C3, C4 = symbols('x,C:5')
```

Only constants in the given range (inclusive) are renumbered; the renumbering always starts from 1:

```
>>> constant_renumber(C1 + C3 + C4, 'C', 1, 3)
C1 + C2 + C4
```

(continues on next page)

(continued from previous page)

```

>>> constant_renumber(C0 + C1 + C3 + C4, 'C', 2, 4)
C0 + 2*C1 + C2
>>> constant_renumber(C0 + 2*C1 + C2, 'C', 0, 1)
C1 + 3*C2
>>> pprint(C2 + C1*x + C3*x**2)
      2
C1*x + C2 + C3*x
>>> pprint(constant_renumber(C2 + C1*x + C3*x**2, 'C', 1, 3))
      2
C1 + C2*x + C3*x

```

constantsimp

`sympy.solvers.ode.constantsimp(expr, constants)`

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with `dsolve()` (page 1230), and is not intended for general use.

Simplification is done by “absorbing” the arbitrary constants into other arbitrary constants, numbers, and symbols that they are not independent of.

The symbols must all have the same name with numbers after it, for example, C_1 , C_2 , C_3 . The `symbolname` here would be ‘C’, the `startnumber` would be 1, and the `endnumber` would be 3. If the arbitrary constants are independent of the variable x , then the independent symbol would be x . There is no need to specify the dependent function, such as $f(x)$, because it already has the independent symbol, x , in it.

Because terms are “absorbed” into arbitrary constants and because constants are renumbered after simplifying, the arbitrary constants in `expr` are not necessarily equal to the ones of the same name in the returned result.

If two or more arbitrary constants are added, multiplied, or raised to the power of each other, they are first absorbed together into a single arbitrary constant. Then the new constant is combined into other terms if necessary.

Absorption of constants is done with limited assistance:

1. terms of `Add` (page 158)s are collected to try join constants so $e^x(C_1 \cos(x) + C_2 \cos(x))$ will simplify to $e^x C_1 \cos(x)$;
2. powers with exponents that are `Add` (page 158)s are expanded so e^{C_1+x} will be simplified to $C_1 e^x$.

Use `constant_renumber()` (page 1239) to renumber constants after simplification or else arbitrary numbers on constants may appear, e.g. $C_1 + C_3 x$.

In rare cases, a single constant can be “simplified” into two constants. Every differential equation solution should have as many arbitrary constants as the order of the differential equation. The result here will be technically correct, but it may, for example, have C_1 and C_2 in an expression, when C_1 is actually equal to C_2 . Use your discretion in such situations, and also take advantage of the ability to use hints in `dsolve()` (page 1230).

Examples

```
>>> from sympy import symbols
>>> from sympy.solvers.ode import constantsimp
>>> C1, C2, C3, x, y = symbols('C1, C2, C3, x, y')
>>> constantsimp(2*C1*x, {C1, C2, C3})
C1*x
>>> constantsimp(C1 + 2 + x, {C1, C2, C3})
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, {C1, C2, C3})
C1 + C3*x
```

sol_simplicity

`sympy.solvers.ode.ode_sol_simplicity(sol, func, trysolving=True)`

Returns an extended integer representing how simple a solution to an ODE is.

The following things are considered, in order from most simple to least:

- `sol` is solved for `func`.
- `sol` is not solved for `func`, but can be if passed to `solve` (e.g., a solution returned by `dsolve(ode, func, simplify=False)`).
- If `sol` is not solved for `func`, then base the result on the length of `sol`, as computed by `len(str(sol))`.
- If `sol` has any unevaluated [Integral](#) (page 690)s, this will automatically be considered less simple than any of the above.

This function returns an integer such that if solution A is simpler than solution B by above metric, then `ode_sol_simplicity(sola, func) < ode_sol_simplicity(solb, func)`.

Currently, the following are the numbers returned, but if the heuristic is ever improved, this may change. Only the ordering is guaranteed.

Simplicity	Return
<code>sol</code> solved for <code>func</code>	-2
<code>sol</code> not solved for <code>func</code> but can be	-1
<code>sol</code> is not solved nor solvable for <code>func</code>	<code>len(str(sol))</code>
<code>sol</code> contains an Integral (page 690)	<code>oo</code>

`oo` here means the SymPy infinity, which should compare greater than any integer.

If you already know `solve()` (page 1296) cannot solve `sol`, you can use `trysolving=False` to skip that step, which is the only potentially slow step. For example, `dsolve()` (page 1230) with the `simplify=False` flag should do this.

If `sol` is a list of solutions, if the worst solution in the list returns `oo` it returns that, otherwise it returns `len(str(sol))`, that is, the length of the string representation of the whole list.

Examples

This function is designed to be passed to `min` as the key argument, such as `min(listofsolutions, key=lambda i: ode_sol_simplicity(i, f(x)))`.

```
>>> from sympy import symbols, Function, Eq, tan, cos, sqrt, Integral
>>> from sympy.solvers.ode import ode_sol_simplicity
>>> x, C1, C2 = symbols('x, C1, C2')
>>> f = Function('f')
```

```
>>> ode_sol_simplicity(Eq(f(x), C1*x**2), f(x))
-2
>>> ode_sol_simplicity(Eq(x**2 + f(x), C1), f(x))
-1
>>> ode_sol_simplicity(Eq(f(x), C1*Integral(2*x, x)), f(x))
oo
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x) + f(x)), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)
```

1st_exact

`sympy.solvers.ode.ode_1st_exact(eq, func, order, match)`

Solves 1st order exact ordinary differential equations.

A 1st order differential equation is called exact if it is the total differential of a function. That is, the differential equation

$$P(x, y) \partial x + Q(x, y) \partial y = 0$$

is exact if there is some function $F(x, y)$ such that $P(x, y) = \partial F / \partial x$ and $Q(x, y) = \partial F / \partial y$. It can be shown that a necessary and sufficient condition for a first order ODE to be exact is that $\partial P / \partial y = \partial Q / \partial x$. Then, the solution will be as given below:

```
>>> from sympy import Function, Eq, Integral, symbols, pprint
>>> x, y, t, x0, y0, C1 = symbols('x,y,t,x0,y0,C1')
>>> P, Q, F = map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
... Integral(Q(x0, t), (t, y0, y))), C1))
```

$$F(x, y) = \int_{x_0}^x P(t, y) dt + \int_{y_0}^y Q(x_0, t) dt = C_1$$

Where the first partials of P and Q exist and are continuous in a simply connected region.

A note: SymPy currently has no way to represent inert substitution on an expression, so the hint `1st_exact_Integral` will return an integral with dy . This is supposed to represent the function that you are solving for.

References

- http://en.wikipedia.org/wiki/Exact_differential_equation

- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 73

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='lst_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

lst_homogeneous_coeff_best

`sympy.solvers.ode.ode_lst_homogeneous_coeff_best(eq, func, order, match)`

Returns the best solution to an ODE from the two hints `lst_homogeneous_coeff_subs_dep_div_indep` and `lst_homogeneous_coeff_subs_indep_div_dep`.

This is as determined by `ode_sol_simplicity()` (page 1241).

See the `ode_lst_homogeneous_coeff_subs_indep_div_dep()` (page 1245) and `ode_lst_homogeneous_coeff_subs_dep_div_indep()` (page 1244) docstrings for more information on these hints. Note that there is no `ode_lst_homogeneous_coeff_best_Integral` hint.

References

- http://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='lst_homogeneous_coeff_best', simplify=False))
          /      2      \
      log| 3*x      |
          |----- + 1|
          | 2        |
          \f (x)    /
log(f(x)) = log(C1) - -----
                        3
```

1st_homogeneous_coeff_subs_dep_div_indep

`sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep(eq, func, order, match)`

Solves a 1st order differential equation with homogeneous coefficients using the substitution $u_1 = \frac{\text{<dependent variable>}}{\text{<independent variable>}}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of `homogeneous_order()` (page 1235).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $y = u_1 x$ (i.e. $u_1 = y/x$) will turn the differential equation into an equation separable in the variables x and u . If $h(u_1)$ is the function that results from making the substitution $u_1 = f(x)/x$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform)
/f(x)\      /f(x)\ d
g|----| + h|----|*--(f(x))
 \ x /      \ x / dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'))
      f(x)
      ----
      x
      /
      |
      |      -h(u1)
      |      ----- d(u1)
      |      u1*h(u1) + g(u1)
log(x) = C1 + |
              /
```

Where $u_1 h(u_1) + g(u_1) \neq 0$ and $x \neq 0$.

See also the docstrings of `ode_1st_homogeneous_coeff_best()` (page 1243) and `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 1245).

References

- http://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 59

indirect doctest

Examples

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep', simplify=False))
```

$$\log(x) = \log(C_1) - \frac{\log\left(\frac{3f(x)}{x} + \frac{f^3(x)}{x}\right)}{3}$$

1st_homogeneous_coeff_subs_indep_div_dep

`sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep(eq, func, order, match)`

Solves a 1st order differential equation with homogeneous coefficients using the substitution $u_2 = \frac{\langle \text{independent variable} \rangle}{\langle \text{dependent variable} \rangle}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of `homogeneous_order()` (page 1235).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $x = u_2 y$ (i.e. $u_2 = x/y$) will turn the differential equation into an equation separable in the variables y and u_2 . If $h(u_2)$ is the function that results from making the substitution $u_2 = x/f(x)$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform)
```

$$g\left(\frac{x}{f(x)}\right) + h\left(\frac{x}{f(x)}\right) \frac{d}{dx} f(x)$$

```
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'))
```

$$\frac{x}{f(x)} \int \frac{-g(u_2)}{u_2^2 g(u_2) + h(u_2)} du_2$$

(continues on next page)

(continued from previous page)

$$f(x) = C1 \cdot e^{\frac{1}{x}}$$

Where $u_2 g(u_2) + h(u_2) \neq 0$ and $f(x) \neq 0$.

See also the docstrings of `ode_1st_homogeneous_coeff_best()` (page 1243) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1244).

References

- http://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 59

indirect doctest

Examples

```
>>> from sympy import Function, pprint, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep',
... simplify=False))
```

$$\log(f(x)) = \log(C1) - \frac{\log\left(\frac{3x^2}{f(x)^2} + 1\right)}{3}$$

1st_linear

`sympy.solvers.ode.ode_1st_linear(eq, func, order, match)`

Solves 1st order linear differential equations.

These are differential equations of the form

$$dy/dx + P(x)y = Q(x).$$

These kinds of differential equations can be solved in a general way. The integrating factor $e^{\int P(x) dx}$ will turn the equation into a separable equation. The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff, sin
>>> from sympy.abc import x
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x))
>>> pprint(genform)
```

(continues on next page)

(continued from previous page)

```

P(x)*f(x) + --(f(x)) = Q(x)
              dx
>>> pprint(dsolve(genform, f(x), hint='1st_linear_Integral'))

```

$$f(x) = C1 + \int Q(x) e^{\int P(x) dx} dx - \int P(x) dx e^{\int P(x) dx}$$

References

- http://en.wikipedia.org/wiki/Linear_differential_equation#First_order_equation
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 92

indirect doctest

Examples

```

>>> f = Function('f')
>>> pprint(dsolve(Eq(x*diff(f(x), x) - f(x), x**2*sin(x)),
... f(x), '1st_linear'))
f(x) = x*(C1 - cos(x))

```

Bernoulli

`sympy.solvers.ode.ode_Bernoulli(eq, func, order, match)`

Solves Bernoulli differential equations.

These are equations of the form

$$dy/dx + P(x)y = Q(x)y^n, n \neq 1.$$

The substitution $w = 1/y^{1-n}$ will transform an equation of this form into one that is linear (see the docstring of `ode_1st_linear()` (page 1246)). The general solution is:

```

>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)
>>> pprint(genform)

```

$$P(x)*f(x) + \frac{d}{dx}(f(x)) = Q(x)*f(x)^n$$

```

>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'))

```

1

(continues on next page)

(continued from previous page)

$$f(x) = \frac{C1 + (-1 + n) \int \frac{-Q(x) e^{\int (1-n) P(x) dx} dx}{e^{\int (-1+n) P(x) dx}}}{1-n}$$

Note that the equation is separable when $n = 1$ (see the docstring of `ode_separable()` (page 1254)).

```
>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x),
... hint='separable_Integral'))
f(x)
|
| 1
| - dy = C1 + | (-P(x) + Q(x)) dx
| y
|
```

References

- http://en.wikipedia.org/wiki/Bernoulli_differential_equation
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 95

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, Eq, pprint, log
>>> from sympy.abc import x
>>> f = Function('f')
```

```
>>> pprint(dsolve(Eq(x*f(x).diff(x) + f(x), log(x)*f(x)**2),
... f(x), hint='Bernoulli'))
f(x) = -----
          /      log(x)  1\
          x*| C1 + ----- + -|
          \      x      x/
```

Liouville

`sympy.solvers.ode.ode_Liouville(eq, func, order, match)`

Solves 2nd order Liouville differential equations.

The general form of a Liouville ODE is

$$\frac{d^2y}{dx^2} + g(y) \left(\frac{dy}{dx}\right)^2 + h(x) \frac{dy}{dx}.$$

The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = Eq(diff(f(x), x, x) + g(f(x))*diff(f(x), x)**2 +
... h(x)*diff(f(x), x), 0)
>>> pprint(genform)
```

$$g(f(x)) \left| \frac{d}{dx} \right|^2 (f(x)) + h(x) \left| \frac{d}{dx} \right| (f(x)) + \frac{d^2}{dx^2} (f(x)) = 0$$

```
>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))
```

$$C1 + C2 \int e^{\int \left(\int h(x) dx \right) dx} dx + \int e^{\int \left(\int g(y) dy \right) dy} dy = 0$$

References

- Goldstein and Braun, “Advanced Methods for the Solution of Differential Equations”, pp. 98
- <http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Liouville>

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(diff(f(x), x, x) + diff(f(x), x)**2/f(x) +
... diff(f(x), x)/x, f(x), hint='Liouville'))
```

$$[f(x) = -\sqrt{C1 + C2 \log(x)}, f(x) = \sqrt{C1 + C2 \log(x)}]$$

Riccati_special_minus2

`sympy.solvers.ode.ode_Riccati_special_minus2(eq, func, order, match)`

The general Riccati equation has the form

$$dy/dx = f(x)y^2 + g(x)y + h(x).$$

While it does not have a general solution [1], the “special” form, $dy/dx = ay^2 - bx^c$, does have solutions in many cases [2]. This routine returns a solution for $a(dy/dx) = by^2 + cy/x + d/x^2$ that is obtained by using a suitable change of variables to reduce it to the special form and is valid when neither a nor b are zero and either c or d is zero.

```
>>> from sympy.abc import x, y, a, b, c, d
>>> from sympy.solvers.ode import dsolve, checkodesol
>>> from sympy import pprint, Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = a*y.diff(x) - (b*y**2 + c*y/x + d/x**2)
>>> sol = dsolve(genform, y)
>>> pprint(sol, wrap_line=False)
```

$$f(x) = \frac{-\sqrt{a + c - \sqrt{4*b*d - (a + c)^2}} \tan\left(C_1 + \frac{\sqrt{4*b*d - (a + c)^2} \log(x)}{2*a}\right)}{2*b*x}$$

```
>>> checkodesol(genform, sol, order=1)[0]
True
```

References

1. <http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Riccati>
2. <http://eqworld.ipmnet.ru/en/solutions/ode/ode0106.pdf> - <http://eqworld.ipmnet.ru/en/solutions/ode/ode0123.pdf>

nth_linear_constant_coeff_homogeneous

`sympy.solvers.ode.ode_nth_linear_constant_coeff_homogeneous(eq, func, order, match, returns='sol')`

Solves an n th order linear homogeneous differential equation with constant coefficients.

This is an equation of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0.$$

These equations can be solved in a general manner, by taking the roots of the characteristic equation $a_n m^n + a_{n-1} m^{n-1} + \cdots + a_1 m + a_0 = 0$. The solution will then be the sum of $C_n x^i e^{rx}$ terms, for each where C_n is an arbitrary constant, r is a root of the characteristic equation and i is one of each from 0 to the multiplicity of the root - 1 (for example, a root 3 of multiplicity 2 would create the terms $C_1 e^{3x} + C_2 x e^{3x}$). The exponential is usually expanded for complex roots using Euler’s equation $e^{Ix} = \cos(x) + I \sin(x)$. Complex roots

always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as $e^{ax} (C_1 \cos(bx) + C_2 \sin(bx))$.

If SymPy cannot find exact roots to the characteristic equation, a `CRootOf` instance will be return instead.

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(f(x).diff(x, 5) + 10*f(x).diff(x) - 2*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous')
...
Eq(f(x), C1*exp(x*CRootOf(_x**5 + 10*_x - 2, 0)) +
C2*exp(x*CRootOf(_x**5 + 10*_x - 2, 1)) +
C3*exp(x*CRootOf(_x**5 + 10*_x - 2, 2)) +
C4*exp(x*CRootOf(_x**5 + 10*_x - 2, 3)) +
C5*exp(x*CRootOf(_x**5 + 10*_x - 2, 4)))
```

Note that because this method does not involve integration, there is no `nth_linear_constant_coeff_homogeneous_Integral` hint.

The following is for internal use:

- `returns = 'sol'` returns the solution to the ODE.
- `returns = 'list'` returns a list of linearly independent solutions, for use with non homogeneous solution methods like variation of parameters and undetermined coefficients. Note that, though the solutions should be linearly independent, this function does not explicitly check that. You can do `assert simplify(wronskian(sollist)) != 0` to check for linear independence. Also, `assert len(sollist) == order` will need to pass.
- `returns = 'both'`, return a dictionary `{'sol': <solution to ODE>, 'list': <list of linearly independent solutions>}`.

References

- http://en.wikipedia.org/wiki/Linear_differential_equation section: Nonhomogeneous_equation_with_constant_coefficients
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 211

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 4) + 2*f(x).diff(x, 3) -
... 2*f(x).diff(x, 2) - 6*f(x).diff(x) + 5*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous'))
...
              x                               -2*x
f(x) = (C1 + C2*x)*e  + (C3*sin(x) + C4*cos(x))*e
```

nth_linear_constant_coeff_undetermined_coefficients

`sympy.solvers.ode.ode_nth_linear_constant_coeff_undetermined_coefficients(eq, func, or-der, match)`

Solves an n th order linear differential equation with constant coefficients using the method of undetermined coefficients.

This method works on differential equations of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x),$$

where $P(x)$ is a function that has a finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form $ax^i e^{bx} \sin(cx + d)$ or $ax^i e^{bx} \cos(cx + d)$, where i is a non-negative integer and a , b , c , and d are constants. For example any polynomial in x , functions like $x^2 e^{2x}$, $x \sin(x)$, and $e^x \cos(x)$ can all be used. Products of sin's and cos's have a finite number of derivatives, because they can be expanded into $\sin(ax)$ and $\cos(bx)$ terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert $\sin^2(x)$ into $(1 + \cos(2x))/2$ to properly apply the method of undetermined coefficients on it.

This method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for each term will be a system of linear equations, which are be solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient x to make them linearly independent.

References

- http://en.wikipedia.org/wiki/Method_of_undetermined_coefficients
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 221

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, pprint, exp, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) + f(x) -
... 4*exp(-x)*x**2 + cos(2*x), f(x),
... hint='nth_linear_constant_coeff_undetermined_coefficients'))
```

$$f(x) = \frac{C1 + C2*x + \frac{x^4}{3}}{e^{-x}} - \frac{4*\sin(2*x)}{25} + \frac{3*\cos(2*x)}{25}$$

nth_linear_constant_coeff_variation_of_parameters

```
sympy.solvers.ode.ode_nth_linear_constant_coeff_variation_of_parameters(eq,
                                                                    func,
                                                                    or-
                                                                    der,
                                                                    match)
```

Solves an n th order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1f'(x) + a_0f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{i=1}^n c_i(x)y_i(x),$$

where y_i is the i th solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by

$$\sum_{i=1}^n \left(\int \frac{W_i(x)}{W(x)} dx \right) y_i(x),$$

where $W(x)$ is the Wronskian of the fundamental system (the system of n linearly independent solutions to the homogeneous equation), and $W_i(x)$ is the Wronskian of the fundamental system with the i th column replaced with $[0, 0, \dots, 0, P(x)]$.

This method is general enough to solve any n th order inhomogeneous linear differential equation with constant coefficients, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the `nth_linear_constant_coeff_variation_of_parameters_Integral` hint and simplifying the integrals manually. Also, prefer using `nth_linear_constant_coeff_undetermined_coefficients` when it applies, because it doesn't use integration, making it faster and more reliable.

Warning, using `simplify=False` with '`nth_linear_constant_coeff_variation_of_parameters`' in `dsolve()` (page 1230) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use `simplify=False` with '`nth_linear_constant_coeff_variation_of_parameters_Integral`' for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

References

- http://en.wikipedia.org/wiki/Variation_of_parameters
- <http://planetmath.org/VariationOfParameters>
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 233

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, pprint, exp, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 3) - 3*f(x).diff(x, 2) +
... 3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
... hint='nth_linear_constant_coeff_variation_of_parameters'))
```

$$f(x) = \left| C1 + C2*x + C3*x^2 + \frac{x^3*(6*\log(x) - 11)}{36} \right| e^x$$

separable

`sympy.solvers.ode.ode_separable(eq, func, order, match)`

Solves separable 1st order differential equations.

This is any differential equation that can be written as $P(y)\frac{dy}{dx} = Q(x)$. The solution can then just be found by rearranging terms and integrating: $\int P(y) dy = \int Q(x) dx$. This hint uses `sympy.simplify.simplify.separatevars()` (page 1157) as its back end, so if a separable equation is not caught by this solver, it is most likely the fault of that function. `separatevars()` (page 1157) is smart enough to do most expansion and factoring necessary to convert a separable equation $F(x, y)$ into the proper form $P(x) \cdot Q(y)$. The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> a, b, c, d, f = map(Function, ['a', 'b', 'c', 'd', 'f'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform)
```

$$a(x)*b(f(x))*\frac{d}{dx}(f(x)) = c(x)*d(f(x))$$

```
>>> pprint(dsolve(genform, f(x), hint='separable_Integral'))
```

$$\int \frac{b(y)}{d(y)} dy = C1 + \int \frac{c(x)}{a(x)} dx$$

References

- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 52

indirect doctest

Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x) + x, 3*x*f(x)**2), f(x),
... hint='separable', simplify=False))
      / 2      \      2
log\3*f (x) - 1/      x
----- = C1 + --
      6              2
```

almost_linear

`sympy.solvers.ode.ode_almost_linear(eq, func, order, match)`

Solves an almost-linear differential equation.

The general form of an almost linear differential equation is

$$f(x)g(y)y + k(x)l(y) + m(x) = 0 \text{ where } l'(y) = g(y).$$

This can be solved by substituting $l(y) = u(y)$. Making the given substitution reduces it to a linear differential equation of the form $u' + P(x)u + Q(x) = 0$.

The general solution is

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, y, n
>>> f, g, k, l = map(Function, ['f', 'g', 'k', 'l'])
>>> genform = Eq(f(x)*(l(y).diff(y)) + k(x)*l(y) + g(x))
>>> pprint(genform)
      d
f(x)*--(l(y)) + g(x) + k(x)*l(y) = 0
      dy
>>> pprint(dsolve(genform, hint = 'almost_linear'))
      /      //      y*k(x)      \
      |      ||      -----      ||
      |      ||      f(x)          ||
      |      ||      -g(x)*e      ||
      |      ||      -----      ||
l(y) = | C1 + <      k(x)          || for k(x) != 0 | -y*k(x)
      |      ||      -y*g(x)      ||      f(x)
      |      ||      -----      ||
      |      ||      f(x)          ||
      |      ||      otherwise      ||
      |      ||      ||
      \      \\\
      \      \\\
```

See also:

`sympy.solvers.ode.ode_1st_linear()` (page 1246)

References

- Joel Moses, "Symbolic Integration - The Stormy Decade", Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

Examples

```
>>> from sympy import Function, Derivative, pprint
>>> from sympy.solvers.ode import dsolve, classify_ode
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), (C1 - Ei(x))*exp(-x))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'))
      -x
f(x) = (C1 - Ei(x))*e
```

linear_coefficients

`sympy.solvers.ode.ode_linear_coefficients(eq, func, order, match)`

Solves a differential equation with linear coefficients.

The general form of a differential equation with linear coefficients is

$$y' + F\left(\frac{a_1x + b_1y + c_1}{a_2x + b_2y + c_2}\right) = 0,$$

where $a_1, b_1, c_1, a_2, b_2, c_2$ are constants and $a_1b_2 - a_2b_1 \neq 0$.

This can be solved by substituting:

$$x = x' + \frac{b_2c_1 - b_1c_2}{a_2b_1 - a_1b_2}$$
$$y = y' + \frac{a_1c_2 - a_2c_1}{a_2b_1 - a_1b_2}.$$

This substitution reduces the equation to a homogeneous differential equation.

See also:

`sympy.solvers.ode.ode_1st_homogeneous_coeff_best()` (page 1243), `sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 1245), `sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1244)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

Examples

```
>>> from sympy import Function, Derivative, pprint
>>> from sympy.solvers.ode import dsolve, classify_ode
>>> from sympy.abc import x
>>> f = Function('f')
```

(continues on next page)

(continued from previous page)

```

>>> df = f(x).diff(x)
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[Eq(f(x), -x - sqrt(C1 + 7*x**2) - 1), Eq(f(x), -x + sqrt(C1 + 7*x**2) - 1)]
>>> pprint(dsolve(eq, hint='linear_coefficients'))

```

$$[f(x) = -x - \sqrt{C1 + 7x^2} - 1, f(x) = -x + \sqrt{C1 + 7x^2} - 1]$$

separable_reduced

`sympy.solvers.ode.ode_separable_reduced(eq, func, order, match)`
 Solves a differential equation that can be reduced to the separable form.

The general form of this equation is

$$y' + (y/x)H(x^n y) = 0.$$

This can be solved by substituting $u(y) = x^n y$. The equation then reduces to the separable form $\frac{u'}{u(\text{power}-H(u))} - \frac{1}{x} = 0$.

The general solution is:

```

>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n
>>> f, g = map(Function, ['f', 'g'])
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform)

```

$$\frac{d}{dx} f(x) + \frac{f(x)^n g(f(x))}{x}$$

```

>>> pprint(dsolve(genform, hint='separable_reduced'))

```

$$\frac{x^n f(x)}{\int \frac{1}{y^{n+1} g(y)} dy} = C1 + \log(x)$$

See also:

[sympy.solvers.ode.ode_separable\(\)](#) (page 1254)

References

- Joel Moses, "Symbolic Integration - The Stormy Decade", Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

Examples

```
>>> from sympy import Function, Derivative, pprint
>>> from sympy.solvers.ode import dsolve, classify_ode
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = (x - x**2*f(x))*d - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (-sqrt(C1*x**2 + 1) + 1)/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/x)]
>>> pprint(dsolve(eq, hint='separable_reduced'))
```

$$[f(x) = \frac{-\sqrt{C1x^2 + 1} + 1}{x}, f(x) = \frac{\sqrt{C1x^2 + 1} + 1}{x}]$$

lie_group

`sympy.solvers.ode.ode_lie_group(eq, func, order, match)`

This hint implements the Lie group method of solving first order differential equations. The aim is to convert the given differential equation from the given coordinate system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE is quadrature and can be solved easily. It makes use of the `sympy.solvers.ode.infinitesimals()` (page 1236) function which returns the infinitesimals of the transformation.

The coordinates r and s can be found by solving the following Partial Differential Equations.

$$\xi \frac{\partial r}{\partial x} + \eta \frac{\partial r}{\partial y} = 0$$

$$\xi \frac{\partial s}{\partial x} + \eta \frac{\partial s}{\partial y} = 1$$

The differential equation becomes separable in the new coordinate system

$$\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x, y) \frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x, y) \frac{\partial r}{\partial y}}$$

After finding the solution by integration, it is then converted back to the original coordinate system by substituting r and s in terms of x and y again.

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

Examples


```
>>> from sympy import Function, dsolve, Eq, exp, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
... hint='lie_group'))
      /      2\      2
      |      x |      -x
f(x) = | C1 + -- | * e
      \      2 /
```

1st_power_series

`sympy.solvers.ode.ode_1st_power_series(eq, func, order, match)`

The power series solution is a method which gives the Taylor series expansion to the solution of a differential equation.

For a first order differential equation $\frac{dy}{dx} = h(x, y)$, a power series solution exists at a point $x = x_0$ if $h(x, y)$ is analytic at x_0 . The solution is given by

$$y(x) = y(x_0) + \sum_{n=1}^{\infty} \frac{F_n(x_0, b)(x - x_0)^n}{n!},$$

where $y(x_0) = b$ is the value of y at the initial value of x_0 . To compute the values of the $F_n(x_0, b)$ the following algorithm is followed, until the required number of terms are generated.

1. $F_1 = h(x_0, b)$
2. $F_{n+1} = \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1$

References

- Travis W. Walker, Analytic power series technique for solving first-order differential equations, p.p 17, 18

Examples

```
>>> from sympy import Function, Derivative, pprint, exp
>>> from sympy.solvers.ode import dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'))
      3      4      5
      C1*x  C1*x  C1*x  / 6\
f(x) = C1 + C1*x - ---- + ---- + ---- + 0\ x /
      6      24      60
```

2nd_power_series_ordinary

`sympy.solvers.ode.ode_2nd_power_series_ordinary(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with

polynomial coefficients at an ordinary point. A homogenous differential equation is of the form

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x) = 0$$

For simplicity it is assumed that $P(x)$, $Q(x)$ and $R(x)$ are polynomials, it is sufficient that $\frac{Q(x)}{P(x)}$ and $\frac{R(x)}{P(x)}$ exists at x_0 . A recurrence relation is obtained by substituting y as $\sum_{n=0}^{\infty} a_n x^n$, in the differential equation, and equating the n th term. Using this relation various terms can be generated.

References

- <http://tutorial.math.lamar.edu/Classes/DE/SeriesSolutions.aspx>
- George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x, y
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'))
```

$$f(x) = C2 \sqrt[4]{x} - \frac{x^2}{2} + 1 + C1 x \sqrt[4]{x} - \frac{x^2}{6} + 1 + 0 \sqrt[6]{x}$$

2nd_power_series_regular

`sympy.solvers.ode.ode_2nd_power_series_regular(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at a regular point. A second order homogenous differential equation is of the form

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x) = 0$$

A point is said to regular singular at x_0 if $x - x_0 \frac{Q(x)}{P(x)}$ and $(x - x_0)^2 \frac{R(x)}{P(x)}$ are analytic at x_0 . For simplicity $P(x)$, $Q(x)$ and $R(x)$ are assumed to be polynomials. The algorithm for finding the power series solutions is:

1. Try expressing $(x - x_0)P(x)$ and $((x - x_0)^2)Q(x)$ as power series solutions about x_0 . Find p_0 and q_0 which are the constants of the power series expansions.
2. Solve the indicial equation $f(m) = m(m - 1) + m * p_0 + q_0$, to obtain the roots m_1 and m_2 of the indicial equation.
3. If $m_1 - m_2$ is a non integer there exists two series solutions. If $m_1 = m_2$, there exists only one solution. If $m_1 - m_2$ is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form $x^m \sum_{n=0}^{\infty} a_n x^n$. The coefficients are determined by the following recurrence relation. $a_n = -\frac{\sum_{k=0}^{n-1} q_{n-k} + (m+k)p_{n-k}}{f(m+n)}$. For the case in which $m_1 - m_2$ is an integer, it can be seen from the recurrence relation that for the lower root m , when n equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.

References

- George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x, y
>>> f = Function("f")
>>> eq = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)
>>> pprint(dsolve(eq))
```

$$f(x) = C2 \sqrt[4]{x} - \frac{x^2}{6} + \frac{1}{120} + \frac{C1 \sqrt[6]{x} - \frac{x^4}{720} + \frac{x^2}{24} - \frac{1}{2}}{x} + 0 \sqrt[6]{x}$$

5.29.3 Lie heuristics

These functions are intended for internal use of the Lie Group Solver. Nonetheless, they contain useful information in their docstrings on the algorithms implemented for the various heuristics.

abaco1_simple

`sympy.solvers.ode.lie_heuristic_abaco1_simple(match, comp=False)`

The first heuristic uses the following four sets of assumptions on ξ and η

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption $\xi = 0$ and η to be a function of x , the PDE

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi * \frac{\partial h}{\partial x} - \eta * \frac{\partial h}{\partial y} = 0$$

reduces to $f'(x) - f \frac{\partial h}{\partial y} = 0$. If $\frac{\partial h}{\partial y}$ is a function of x , then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco1_product

`sympy.solvers.ode.lie_heuristic_abaco1_product(match, comp=False)`

The second heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) * g(y)$$

$$\eta = f(x) * g(y), \xi = 0$$

The first assumption of this heuristic holds good if $\frac{1}{h^2} \frac{\partial^2}{\partial x \partial y} \log(h)$ is separable in x and y , then the separated factors containing x is $f(x)$, and $g(y)$ is obtained by

$$e^{\int f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right) dy}$$

provided $f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right)$ is a function of y only.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) * g(y)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

bivariate

`sympy.solvers.ode.lie_heuristic_bivariate(match, comp=False)`

The third heuristic assumes the infinitesimals ξ and η to be bi-variate polynomials in x and y . The assumption made here for the logic below is that h is a rational function in x and y though that may not be necessary for the infinitesimals to be bivariate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

References

- Lie Groups and Differential Equations pp. 327 - pp. 329

chi

`sympy.solvers.ode.lie_heuristic_chi(match, comp=False)`

The aim of the fourth heuristic is to find the function $\chi(x, y)$ that satisfies the PDE $\frac{d\chi}{dx} + h\frac{\partial\chi}{\partial y} - \frac{\partial h}{\partial y}\chi = 0$.

This assumes χ to be a bivariate polynomial in x and y . By intuition, h should be a rational function in x and y . The method used here is to substitute a general binomial for χ up to a certain maximum degree is reached. The coefficients of the polynomials, are calculated by by collecting terms of the same order in x and y .

After finding χ , the next step is to use $\eta = \xi * h + \chi$, to determine ξ and η . This can be done by dividing χ by h which would give $-\xi$ as the quotient and η as the remainder.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco2_similar

`sympy.solvers.ode.lie_heuristic_abaco2_similar(match, comp=False)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = g(x), \xi = f(x)$$

$$\eta = f(y), \xi = g(y)$$

For the first assumption,

1. First $\frac{\frac{\partial h}{\partial y}}{\frac{\partial^2 h}{\partial y^2}}$ is calculated. Let us say this value is A
2. If this is constant, then h is matched to the form $A(x) + B(x)e^{\frac{y}{C}}$ then, $\frac{e^{\int \frac{A(x)}{C} dx}}{B(x)}$ gives $f(x)$ and $A(x) * f(x)$ gives $g(x)$
3. Otherwise $\frac{\frac{\partial A}{\partial y}}{\frac{\partial A}{\partial x}} = \gamma$ is calculated. If
 - a] γ is a function of x alone
 - b] $\frac{\gamma \frac{\partial h}{\partial y} - \gamma'(x) - \frac{\partial h}{\partial x}}{h + \gamma} = G$ is a function of x alone. then, $e^{\int G dx}$ gives $f(x)$ and $-\gamma * f(x)$ gives $g(x)$

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(x)$, the coordinates are again interchanged, to get ξ as $f(x^*)$ and η as $g(y^*)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

function_sum

`sympy.solvers.ode.lie_heuristic_function_sum(match, comp=False)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) + g(y)$$

$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y} \left[\left(h \frac{\partial^2}{\partial x^2} (h^{-1}) \right)^{-1} \right]$$

is separable in x and y ,

1. The separated factors containing y is $\frac{\partial g}{\partial y}$. From this $g(y)$ can be determined.
2. The separated factors containing x is $f''(x)$.
3. $h \frac{\partial^2}{\partial x^2} (h^{-1})$ equals $\frac{f''(x)}{f(x)+g(y)}$. From this $f(x)$ can be determined.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) + g(y)$.

For both assumptions, the constant factors are separated among $g(y)$ and $f''(x)$, such that $f''(x)$ obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

abaco2_unique_unknown

`sympy.solvers.ode.lie_heuristic_abaco2_unique_unknown(match, comp=False)`

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1. A list of all functions and non-integer powers containing x and y
2. Loop over each element f in the list, find $\frac{\partial f}{\partial x} = R$

If it is separable in x and y , let X be the factors containing x . Then

a] Check if $\xi = X$ and $\eta = -\frac{X}{R}$ satisfy the PDE. If yes, then return ξ and η

b] Check if $\xi = \frac{-R}{X}$ and $\eta = -\frac{1}{X}$ satisfy the PDE. If yes, then return ξ and η

If not, then check if

a] $\xi = -R, \eta = 1$

b] $\xi = 1, \eta = -\frac{1}{R}$

are solutions.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

abaco2_unique_general

`sympy.solvers.ode.lie_heuristic_abaco2_unique_general(match, comp=False)`

This heuristic finds if infinitesimals of the form $\eta = f(x)$, $\xi = g(y)$ without making any assumptions on h .

The complete sequence of steps is given in the paper mentioned below.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

linear

`sympy.solvers.ode.lie_heuristic_linear(match, comp=False)`

This heuristic assumes

1. $\xi = ax + by + c$ and
2. $\eta = fx + gy + h$

After substituting the following assumptions in the determining PDE, it reduces to

$$f + (g - a)h - bh^2 - (ax + by + c)\frac{\partial h}{\partial x} - (fx + gy + c)\frac{\partial h}{\partial y}$$

Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that h need not be a rational function in this case.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

5.29.4 System of ODEs

These functions are intended for internal use by `dsolve()` (page 1230) for system of differential equations.

system_of_odes_linear_2eq_order1_type1

`sympy.solvers.ode._linear_2eq_order1_type1(x, y, t, r, eq)`

It is classified under system of two linear homogeneous first-order constant-coefficient ordinary differential equations.

The equations which come under this type are

$$x' = ax + by,$$

$$y' = cx + dy$$

The characteristics equation is written as

$$\lambda^2 + (a + d)\lambda + ad - bc = 0$$

and its discriminant is $D = (a - d)^2 + 4bc$. There are several cases

1. Case when $ad - bc \neq 0$. The origin of coordinates, $x = y = 0$, is the only stationary point; it is - a node if $D = 0$ - a node if $D > 0$ and $ad - bc > 0$ - a saddle if $D > 0$ and $ad - bc < 0$ - a focus if $D < 0$ and $a + d \neq 0$ - a centre if $D < 0$ and $a + d = 0$.

1.1. If $D > 0$. The characteristic equation has two distinct real roots λ_1 and λ_2 . The general solution of the system in question is expressed as

$$x = C_1 b e^{\lambda_1 t} + C_2 b e^{\lambda_2 t}$$

$$y = C_1 (\lambda_1 - a) e^{\lambda_1 t} + C_2 (\lambda_2 - a) e^{\lambda_2 t}$$

where C_1 and C_2 being arbitrary constants

1.2. If $D < 0$. The characteristics equation has two conjugate roots, $\lambda_1 = \sigma + i\beta$ and $\lambda_2 = \sigma - i\beta$. The general solution of the system is given by

$$x = b e^{\sigma t} (C_1 \sin(\beta t) + C_2 \cos(\beta t))$$

$$y = e^{\sigma t} [((\sigma - a)C_1 - \beta C_2] \sin(\beta t) + [\beta C_1 + (\sigma - a)C_2 \cos(\beta t)]$$

1.3. If $D = 0$ and $a \neq d$. The characteristic equation has two equal roots, $\lambda_1 = \lambda_2$. The general solution of the system is written as

$$x = 2b(C_1 + \frac{C_2}{a-d} + C_2 t) e^{\frac{a+d}{2}t}$$

$$y = [(d-a)C_1 + C_2 + (d-a)C_2 t] e^{\frac{a+d}{2}t}$$

1.4. If $D = 0$ and $a = d \neq 0$ and $b = 0$

$$x = C_1 e^{at}, y = (cC_1 t + C_2) e^{at}$$

1.5. If $D = 0$ and $a = d \neq 0$ and $c = 0$

$$x = (bC_1 t + C_2) e^{at}, y = C_1 e^{at}$$

2. Case when $ad - bc = 0$ and $a^2 + b^2 > 0$. The whole straight line $ax + by = 0$ consists of singular points. The original system of differential equations can be rewritten as

$$x' = ax + by, y' = k(ax + by)$$

2.1 If $a + bk \neq 0$, solution will be

$$x = bC_1 + C_2 e^{(a+bk)t}, y = -aC_1 + kC_2 e^{(a+bk)t}$$

2.2 If $a + bk = 0$, solution will be

$$x = C_1 (bkt - 1) + bC_2 t, y = k^2 bC_1 t + (bk^2 t + 1)C_2$$

system_of_odes_linear_2eq_order1_type2

`sympy.solvers.ode._linear_2eq_order1_type2(x, y, t, r, eq)`

The equations of this type are

$$x' = ax + by + k1, y' = cx + dy + k2$$

The general solution of this system is given by sum of its particular solution and the general solution of the corresponding homogeneous system is obtained from type1.

1. When $ad - bc \neq 0$. The particular solution will be $x = x_0$ and $y = y_0$ where x_0 and y_0 are determined by solving linear system of equations

$$ax_0 + by_0 + k1 = 0, cx_0 + dy_0 + k2 = 0$$

2. When $ad - bc = 0$ and $a^2 + b^2 > 0$. In this case, the system of equation becomes

$$x' = ax + by + k1, y' = k(ax + by) + k2$$

2.1 If $\sigma = a + bk \neq 0$, particular solution is given by

$$x = b\sigma^{-1}(c_1k - c_2)t - \sigma^{-2}(ac_1 + bc_2)$$

$$y = kx + (c_2 - c_1k)t$$

2.2 If $\sigma = a + bk = 0$, particular solution is given by

$$x = \frac{1}{2}b(c_2 - c_1k)t^2 + c_1t$$

$$y = kx + (c_2 - c_1k)t$$

system_of_odes_linear_2eq_order1_type3

`sympy.solvers.ode._linear_2eq_order1_type3(x, y, t, r, eq)`

The equations of this type of ode are

$$x' = f(t)x + g(t)y$$

$$y' = g(t)x + f(t)y$$

The solution of such equations is given by

$$x = e^F(C_1e^G + C_2e^{-G}), y = e^F(C_1e^G - C_2e^{-G})$$

where C_1 and C_2 are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

system_of_odes_linear_2eq_order1_type4

`sympy.solvers.ode._linear_2eq_order1_type4(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = -g(t)x + f(t)y$$

The solution is given by

$$x = F(C_1 \cos(G) + C_2 \sin(G)), y = F(-C_1 \sin(G) + C_2 \cos(G))$$

where C_1 and C_2 are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

system_of_odes_linear_2eq_order1_type5

`sympy.solvers.ode._linear_2eq_order1_type5(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = ag(t)x + [f(t) + bg(t)]y$$

The transformation of

$$x = e^{\int f(t) dt} u, y = e^{\int f(t) dt} v, T = \int g(t) dt$$

leads to a system of constant coefficient linear differential equations

$$u'(T) = v, v'(T) = au + bv$$

system_of_odes_linear_2eq_order1_type6

`sympy.solvers.ode._linear_2eq_order1_type6(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = a[f(t) + ah(t)]x + a[g(t) - h(t)]y$$

This is solved by first multiplying the first equation by $-a$ and adding it to the second equation to obtain

$$y' - ax' = -ah(t)(y - ax)$$

Setting $U = y - ax$ and integrating the equation we arrive at

$$y - ax = C_1 e^{-a \int h(t) dt}$$

and on substituting the value of y in first equation give rise to first order ODEs. After solving for x , we can obtain y by substituting the value of x in second equation.

system_of_odes_linear_2eq_order1_type7

`sympy.solvers.ode._linear_2eq_order1_type7(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = h(t)x + p(t)y$$

Differentiating the first equation and substituting the value of y from second equation will give a second-order linear equation

$$gx'' - (fg + gp + g')x' + (fgp - g^2h + fg' - f'g)x = 0$$

This above equation can be easily integrated if following conditions are satisfied.

1. $fgp - g^2h + fg' - f'g = 0$
2. $fgp - g^2h + fg' - f'g = ag, fg + gp + g' = bg$

If first condition is satisfied then it is solved by current dsolve solver and in second case it becomes a constant coefficient differential equation which is also solved by current solver.

Otherwise if the above condition fails then, a particular solution is assumed as $x = x_0(t)$ and $y = y_0(t)$ Then the general solution is expressed as

$$x = C_1x_0(t) + C_2x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt$$

$$y = C_1y_0(t) + C_2\left[\frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt\right]$$

where C1 and C2 are arbitrary constants and

$$F(t) = e^{\int f(t) dt}, P(t) = e^{\int p(t) dt}$$

system_of_odes_linear_2eq_order2_type1

`sympy.solvers.ode._linear_2eq_order2_type1(x, y, t, r, eq)`

System of two constant-coefficient second-order linear homogeneous differential equations

$$x'' = ax + by$$

$$y'' = cx + dy$$

The characteristic equation for above equations

$$\lambda^4 - (a+d)\lambda^2 + ad - bc = 0$$

whose discriminant is $D = (a-d)^2 + 4bc \neq 0$

1. When $ad - bc \neq 0$

1.1. If $D \neq 0$. The characteristic equation has four distinct roots, $\lambda_1, \lambda_2, \lambda_3, \lambda_4$. The general solution of the system is

$$x = C_1 b e^{\lambda_1 t} + C_2 b e^{\lambda_2 t} + C_3 b e^{\lambda_3 t} + C_4 b e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 - a)e^{\lambda_1 t} + C_2(\lambda_2^2 - a)e^{\lambda_2 t} + C_3(\lambda_3^2 - a)e^{\lambda_3 t} + C_4(\lambda_4^2 - a)e^{\lambda_4 t}$$

where $C_1, \dots, C_4$ are arbitrary constants.

1.2. If $D = 0$ and $a \neq d$:

$$x = 2C_1\left(bt + \frac{2bk}{a-d}\right)e^{\frac{kt}{2}} + 2C_2\left(bt + \frac{2bk}{a-d}\right)e^{-\frac{kt}{2}} + 2bC_3te^{\frac{kt}{2}} + 2bC_4te^{-\frac{kt}{2}}$$

$$y = C_1(d-a)te^{\frac{kt}{2}} + C_2(d-a)te^{-\frac{kt}{2}} + C_3[(d-a)t + 2k]e^{\frac{kt}{2}} + C_4[(d-a)t - 2k]e^{-\frac{kt}{2}}$$

where $C_1, \dots, C_4$ are arbitrary constants and $k = \sqrt{2(a+d)}$

1.3. If $D = 0$ and $a = d \neq 0$ and $b = 0$:

$$x = 2\sqrt{a}C_1e^{\sqrt{a}t} + 2\sqrt{a}C_2e^{-\sqrt{a}t}$$

$$y = cC_1te^{\sqrt{a}t} - cC_2te^{-\sqrt{a}t} + C_3e^{\sqrt{a}t} + C_4e^{-\sqrt{a}t}$$

1.4. If $D = 0$ and $a = d \neq 0$ and $c = 0$:

$$x = bC_1te^{\sqrt{a}t} - bC_2te^{-\sqrt{a}t} + C_3e^{\sqrt{a}t} + C_4e^{-\sqrt{a}t}$$

$$y = 2\sqrt{a}C_1e^{\sqrt{a}t} + 2\sqrt{a}C_2e^{-\sqrt{a}t}$$

2. When $ad - bc = 0$ and $a^2 + b^2 > 0$. Then the original system becomes

$$x'' = ax + by$$

$$y'' = k(ax + by)$$

2.1. If $a + bk \neq 0$:

$$x = C_1e^{t\sqrt{a+bk}} + C_2e^{-t\sqrt{a+bk}} + C_3bt + C_4b$$

$$y = C_1ke^{t\sqrt{a+bk}} + C_2ke^{-t\sqrt{a+bk}} - C_3at - C_4a$$

2.2. If $a + bk = 0$:

$$x = C_1bt^3 + C_2bt^2 + C_3t + C_4$$

$$y = kx + 6C_1t + 2C_2$$

system_of_odes_linear_2eq_order2_type2

`sympy.solvers.ode._linear_2eq_order2_type2(x, y, t, r, eq)`

The equations in this type are

$$x'' = a_1x + b_1y + c_1$$

$$y'' = a_2x + b_2y + c_2$$

The general solution of this system is given by the sum of its particular solution and the general solution of the homogeneous system. The general solution is given by the linear system of 2 equation of order 2 and type 1

1. If $a_1b_2 - a_2b_1 \neq 0$. A particular solution will be $x = x_0$ and $y = y_0$ where the constants x_0 and y_0 are determined by solving the linear algebraic system

$$a_1x_0 + b_1y_0 + c_1 = 0, a_2x_0 + b_2y_0 + c_2 = 0$$

2. If $a_1b_2 - a_2b_1 = 0$ and $a_1^2 + b_1^2 > 0$. In this case, the system in question becomes

$$x'' = ax + by + c_1, y'' = k(ax + by) + c_2$$

2.1. If $\sigma = a + bk \neq 0$, the particular solution will be

$$x = \frac{1}{2}b\sigma^{-1}(c_1k - c_2)t^2 - \sigma^{-2}(ac_1 + bc_2)$$

$$y = kx + \frac{1}{2}(c_2 - c_1k)t^2$$

2.2. If $\sigma = a + bk = 0$, the particular solution will be

$$x = \frac{1}{24}b(c_2 - c_1k)t^4 + \frac{1}{2}c_1t^2$$

$$y = kx + \frac{1}{2}(c_2 - c_1k)t^2$$

system_of_odes_linear_2eq_order2_type3

`sympy.solvers.ode._linear_2eq_order2_type3(x, y, t, r, eq)`

These type of equation is used for describing the horizontal motion of a pendulum taking into account the Earth rotation. The solution is given with $a^2 + 4b > 0$:

$$x = C_1 \cos(\alpha t) + C_2 \sin(\alpha t) + C_3 \cos(\beta t) + C_4 \sin(\beta t)$$

$$y = -C_1 \sin(\alpha t) + C_2 \cos(\alpha t) - C_3 \sin(\beta t) + C_4 \cos(\beta t)$$

where $C_1, \dots, C_4$ and

$$\alpha = \frac{1}{2}a + \frac{1}{2}\sqrt{a^2 + 4b}, \beta = \frac{1}{2}a - \frac{1}{2}\sqrt{a^2 + 4b}$$

system_of_odes_linear_2eq_order2_type4

`sympy.solvers.ode._linear_2eq_order2_type4(x, y, t, r, eq)`

These equations are found in the theory of oscillations

$$x'' + a_1x' + b_1y' + c_1x + d_1y = k_1e^{i\omega t}$$

$$y'' + a_2x' + b_2y' + c_2x + d_2y = k_2e^{i\omega t}$$

The general solution of this linear nonhomogeneous system of constant-coefficient differential equations is given by the sum of its particular solution and the general solution of the corresponding homogeneous system (with $k_1 = k_2 = 0$)

1. A particular solution is obtained by the method of undetermined coefficients:

$$x = A_* e^{i\omega t}, y = B_* e^{i\omega t}$$

On substituting these expressions into the original system of differential equations, one arrive at a linear nonhomogeneous system of algebraic equations for the coefficients A and B .

2. The general solution of the homogeneous system of differential equations is determined by a linear combination of linearly independent particular solutions determined by the method of undetermined coefficients in the form of exponentials:

$$x = Ae^{\lambda t}, y = Be^{\lambda t}$$

On substituting these expressions into the original system and collecting the coefficients of the unknown A and B , one obtains

$$(\lambda^2 + a_1\lambda + c_1)A + (b_1\lambda + d_1)B = 0$$

$$(a_2\lambda + c_2)A + (\lambda^2 + b_2\lambda + d_2)B = 0$$

The determinant of this system must vanish for nontrivial solutions A, B to exist. This requirement results in the following characteristic equation for λ

$$(\lambda^2 + a_1\lambda + c_1)(\lambda^2 + b_2\lambda + d_2) - (b_1\lambda + d_1)(a_2\lambda + c_2) = 0$$

If all roots $k_1, \dots, k_4$ of this equation are distinct, the general solution of the original system of the differential equations has the form

$$x = C_1(b_1\lambda_1 + d_1)e^{\lambda_1 t} - C_2(b_1\lambda_2 + d_1)e^{\lambda_2 t} - C_3(b_1\lambda_3 + d_1)e^{\lambda_3 t} - C_4(b_1\lambda_4 + d_1)e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 + a_1\lambda_1 + c_1)e^{\lambda_1 t} + C_2(\lambda_2^2 + a_1\lambda_2 + c_1)e^{\lambda_2 t} + C_3(\lambda_3^2 + a_1\lambda_3 + c_1)e^{\lambda_3 t} + C_4(\lambda_4^2 + a_1\lambda_4 + c_1)e^{\lambda_4 t}$$

`system_of_odes_linear_2eq_order2_type5`

`sympy.solvers.ode._linear_2eq_order2_type5(x, y, t, r, eq)`

The equation which come under this category are

$$x'' = a(ty' - y)$$

$$y'' = b(tx' - x)$$

The transformation

$$u = tx' - x, b = ty' - y$$

leads to the first-order system

$$u' = atv, v' = btu$$

The general solution of this system is given by

If $ab > 0$:

$$u = C_1 a e^{\frac{1}{2}\sqrt{ab}t^2} + C_2 a e^{-\frac{1}{2}\sqrt{ab}t^2}$$

$$v = C_1 \sqrt{ab} e^{\frac{1}{2}\sqrt{ab}t^2} - C_2 \sqrt{ab} e^{-\frac{1}{2}\sqrt{ab}t^2}$$

If $ab < 0$:

$$u = C_1 a \cos\left(\frac{1}{2}\sqrt{|ab|}t^2\right) + C_2 a \sin\left(-\frac{1}{2}\sqrt{|ab|}t^2\right)$$

$$v = C_1 \sqrt{|ab|} \sin\left(\frac{1}{2}\sqrt{|ab|}t^2\right) + C_2 \sqrt{|ab|} \cos\left(-\frac{1}{2}\sqrt{|ab|}t^2\right)$$

where C_1 and C_2 are arbitrary constants. On substituting the value of u and v in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{v}{t^2} dt$$

where C_3 and C_4 are arbitrary constants.

system_of_odes_linear_2eq_order2_type6

`sympy.solvers.ode._linear_2eq_order2_type6(x, y, t, r, eq)`

The equations are

$$x'' = f(t)(a_1 x + b_1 y)$$

$$y'' = f(t)(a_2 x + b_2 y)$$

If k_1 and k_2 are roots of the quadratic equation

$$k^2 - (a_1 + b_2)k + a_1 b_2 - a_2 b_1 = 0$$

Then by multiplying appropriate constants and adding together original equations we obtain two independent equations:

$$z_1'' = k_1 f(t) z_1, z_1 = a_2 x + (k_1 - a_1) y$$

$$z_2'' = k_2 f(t) z_2, z_2 = a_2 x + (k_2 - a_1) y$$

Solving the equations will give the values of x and y after obtaining the value of z_1 and z_2 by solving the differential equation and substituting the result.

system_of_odes_linear_2eq_order2_type7

`sympy.solvers.ode._linear_2eq_order2_type7(x, y, t, r, eq)`

The equations are given as

$$x'' = f(t)(a_1 x' + b_1 y')$$

$$y'' = f(t)(a_2 x' + b_2 y')$$

If k_1 and ' k_2 ' are roots of the quadratic equation

$$k^2 - (a_1 + b_2)k + a_1 b_2 - a_2 b_1 = 0$$

Then the system can be reduced by adding together the two equations multiplied by appropriate constants give following two independent equations:

$$z_1'' = k_1 f(t) z_1', z_1 = a_2 x + (k_1 - a_1) y$$

$$z_2'' = k_2 f(t) z_2', z_2 = a_2 x + (k_2 - a_1) y$$

Integrating these and returning to the original variables, one arrives at a linear algebraic system for the unknowns x and y :

$$a_2 x + (k_1 - a_1) y = C_1 \int e^{k_1 F(t)} dt + C_2$$

$$a_2 x + (k_2 - a_1) y = C_3 \int e^{k_2 F(t)} dt + C_4$$

where $C_1, \dots, C_4$ are arbitrary constants and $F(t) = \int f(t) dt$

system_of_odes_linear_2eq_order2_type8

`sympy.solvers.ode._linear_2eq_order2_type8(x, y, t, r, eq)`

The equation of this category are

$$x'' = af(t)(ty' - y)$$

$$y'' = bf(t)(tx' - x)$$

The transformation

$$u = tx' - x, v = ty' - y$$

leads to the system of first-order equations

$$u' = atf(t)v, v' = btf(t)u$$

The general solution of this system has the form

If $ab > 0$:

$$u = C_1 a e^{\sqrt{ab} \int tf(t) dt} + C_2 a e^{-\sqrt{ab} \int tf(t) dt}$$

$$v = C_1 \sqrt{ab} e^{\sqrt{ab} \int tf(t) dt} - C_2 \sqrt{ab} e^{-\sqrt{ab} \int tf(t) dt}$$

If $ab < 0$:

$$u = C_1 a \cos(\sqrt{|ab|} \int tf(t) dt) + C_2 a \sin(-\sqrt{|ab|} \int tf(t) dt)$$

$$v = C_1 \sqrt{|ab|} \sin(\sqrt{|ab|} \int tf(t) dt) + C_2 \sqrt{|ab|} \cos(-\sqrt{|ab|} \int tf(t) dt)$$

where C_1 and C_2 are arbitrary constants. On substituting the value of u and v in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{v}{t^2} dt$$

where C_3 and C_4 are arbitrary constants.

system_of_odes_linear_2eq_order2_type9

`sympy.solvers.ode._linear_2eq_order2_type9(x, y, t, r, eq)`

$$t^2 x'' + a_1 t x' + b_1 t y' + c_1 x + d_1 y = 0$$

$$t^2 y'' + a_2 t x' + b_2 t y' + c_2 x + d_2 y = 0$$

These system of equations are euler type.

The substitution of $t = \sigma e^\tau$ ($\sigma \neq 0$) leads to the system of constant coefficient linear differential equations

$$x'' + (a_1 - 1)x' + b_1 y' + c_1 x + d_1 y = 0$$

$$y'' + a_2 x' + (b_2 - 1)y' + c_2 x + d_2 y = 0$$

The general solution of the homogeneous system of differential equations is determined by a linear combination of linearly independent particular solutions determined by the method of undetermined coefficients in the form of exponentials

$$x = Ae^{\lambda t}, y = Be^{\lambda t}$$

On substituting these expressions into the original system and collecting the coefficients of the unknown A and B , one obtains

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)A + (b_1\lambda + d_1)B = 0$$

$$(a_2\lambda + c_2)A + (\lambda^2 + (b_2 - 1)\lambda + d_2)B = 0$$

The determinant of this system must vanish for nontrivial solutions A, B to exist. This requirement results in the following characteristic equation for λ

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)(\lambda^2 + (b_2 - 1)\lambda + d_2) - (b_1\lambda + d_1)(a_2\lambda + c_2) = 0$$

If all roots $k_1, \dots, k_4$ of this equation are distinct, the general solution of the original system of the differential equations has the form

$$x = C_1(b_1\lambda_1 + d_1)e^{\lambda_1 t} - C_2(b_1\lambda_2 + d_1)e^{\lambda_2 t} - C_3(b_1\lambda_3 + d_1)e^{\lambda_3 t} - C_4(b_1\lambda_4 + d_1)e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 + (a_1 - 1)\lambda_1 + c_1)e^{\lambda_1 t} + C_2(\lambda_2^2 + (a_1 - 1)\lambda_2 + c_1)e^{\lambda_2 t} + C_3(\lambda_3^2 + (a_1 - 1)\lambda_3 + c_1)e^{\lambda_3 t} + C_4(\lambda_4^2 + (a_1 - 1)\lambda_4 + c_1)e^{\lambda_4 t}$$

system_of_odes_linear_2eq_order2_type10

`sympy.solvers.ode._linear_2eq_order2_type10(x, y, t, r, eq)`

The equation of this category are

$$(\alpha t^2 + \beta t + \gamma)^2 x'' = ax + by$$

$$(\alpha t^2 + \beta t + \gamma)^2 y'' = cx + dy$$

The transformation

$$\tau = \int \frac{1}{\alpha t^2 + \beta t + \gamma} dt, u = \frac{x}{\sqrt{|\alpha t^2 + \beta t + \gamma|}}, v = \frac{y}{\sqrt{|\alpha t^2 + \beta t + \gamma|}}$$

leads to a constant coefficient linear system of equations

$$u'' = (a - \alpha\gamma + \frac{1}{4}\beta^2)u + bv$$

$$v'' = cu + (d - \alpha\gamma + \frac{1}{4}\beta^2)v$$

These system of equations obtained can be solved by type1 of System of two constant-coefficient second-order linear homogeneous differential equations.

`system_of_odes_linear_2eq_order2_type1`

`sympy.solvers.ode._linear_2eq_order2_type1(x, y, t, r, eq)`

The equations which comes under this type are

$$x'' = f(t)(tx' - x) + g(t)(ty' - y)$$

$$y'' = h(t)(tx' - x) + p(t)(ty' - y)$$

The transformation

$$u = tx' - x, v = ty' - y$$

leads to the linear system of first-order equations

$$u' = tf(t)u + tg(t)v, v' = th(t)u + tp(t)v$$

On substituting the value of u and v in transformed equation gives value of x and y as

$$x = C_3t + t \int \frac{u}{t^2} dt, y = C_4t + t \int \frac{v}{t^2} dt.$$

where C_3 and C_4 are arbitrary constants.

`system_of_odes_linear_3eq_order1_type1`

`sympy.solvers.ode._linear_3eq_order1_type1(x, y, z, t, r, eq)`

$$x' = ax$$

$$y' = bx + cy$$

$$z' = dx + ky + pz$$

Solution of such equations are forward substitution. Solving first equations gives the value of x , substituting it in second and third equation and solving second equation gives y and similarly substituting y in third equation give z .

$$x = C_1e^{at}$$

$$y = \frac{bC_1}{a-c}e^{at} + C_2e^{ct}$$

$$z = \frac{C_1}{a-p}(d + \frac{bk}{a-c})e^{at} + \frac{kC_2}{c-p}e^{ct} + C_3e^{pt}$$

where C_1, C_2 and C_3 are arbitrary constants.

system_of_odes_linear_3eq_order1_type2

`sympy.solvers.ode._linear_3eq_order1_type2(x, y, z, t, r, eq)`

The equations of this type are

$$x' = cy - bz$$

$$y' = az - cx$$

$$z' = bx - ay$$

1. First integral:

$$ax + by + cz = A \quad - (1)$$

$$x^2 + y^2 + z^2 = B^2 \quad - (2)$$

where A and B are arbitrary constants. It follows from these integrals that the integral lines are circles formed by the intersection of the planes (1) and sphere (2)

2. Solution:

$$x = aC_0 + kC_1 \cos(kt) + (cC_2 - bC_3) \sin(kt)$$

$$y = bC_0 + kC_2 \cos(kt) + (aC_2 - cC_3) \sin(kt)$$

$$z = cC_0 + kC_3 \cos(kt) + (bC_2 - aC_3) \sin(kt)$$

where $k = \sqrt{a^2 + b^2 + c^2}$ and the four constants of integration, $C_1, \dots, C_4$ are constrained by a single relation,

$$aC_1 + bC_2 + cC_3 = 0$$

system_of_odes_linear_3eq_order1_type3

`sympy.solvers.ode._linear_3eq_order1_type3(x, y, z, t, r, eq)`

Equations of this system of ODEs

$$ax' = bc(y - z)$$

$$by' = ac(z - x)$$

$$cz' = ab(x - y)$$

1. First integral:

$$a^2x + b^2y + c^2z = A$$

where A is an arbitrary constant. It follows that the integral lines are plane curves.

2. Solution:

$$x = C_0 + kC_1 \cos(kt) + a^{-1}bc(C_2 - C_3) \sin(kt)$$

$$y = C_0 + kC_2 \cos(kt) + ab^{-1}c(C_3 - C_1) \sin(kt)$$

$$z = C_0 + kC_3 \cos(kt) + abc^{-1}(C_1 - C_2) \sin(kt)$$

where $k = \sqrt{a^2 + b^2 + c^2}$ and the four constants of integration, $C_1, \dots, C_4$ are constrained by a single relation

$$a^2C_1 + b^2C_2 + c^2C_3 = 0$$

system_of_odes_linear_3eq_order1_type4

`sympy.solvers.ode._linear_3eq_order1_type4(x, y, z, t, r, eq)`

Equations:

$$x' = (a_1 f(t) + g(t))x + a_2 f(t)y + a_3 f(t)z$$

$$y' = b_1 f(t)x + (b_2 f(t) + g(t))y + b_3 f(t)z$$

$$z' = c_1 f(t)x + c_2 f(t)y + (c_3 f(t) + g(t))z$$

The transformation

$$x = e^{\int g(t) dt} u, y = e^{\int g(t) dt} v, z = e^{\int g(t) dt} w, \tau = \int f(t) dt$$

leads to the system of constant coefficient linear differential equations

$$u' = a_1 u + a_2 v + a_3 w$$

$$v' = b_1 u + b_2 v + b_3 w$$

$$w' = c_1 u + c_2 v + c_3 w$$

These system of equations are solved by homogeneous linear system of constant coefficients of n equations of first order. Then substituting the value of u, v and w in transformed equation gives value of x, y and z .

system_of_odes_linear_neq_order1_type1

`sympy.solvers.ode._linear_neq_order1_type1(match_)`

System of n first-order constant-coefficient linear nonhomogeneous differential equation

$$y'_k = a_{k1}y_1 + a_{k2}y_2 + \dots + a_{kn}y_n; k = 1, 2, \dots, n$$

or that can be written as $\vec{y}' = A \cdot \vec{y}$ where $\vec{y}$ is matrix of y_k for $k = 1, 2, \dots, n$ and A is a $n \times n$ matrix.

Since these equations are equivalent to a first order homogeneous linear differential equation. So the general solution will contain n linearly independent parts and solution will consist some type of exponential functions. Assuming $y = \vec{v}e^{rt}$ is a solution of the system where $\vec{v}$ is a vector of coefficients of $y_1, \dots, y_n$. Substituting y and $y' = r\vec{v}e^{rt}$ into the equation $\vec{y}' = A \cdot \vec{y}$, we get

$$r\vec{v}e^{rt} = A\vec{v}e^{rt}$$

$$r\vec{v} = A\vec{v}$$

where r comes out to be eigenvalue of A and vector $\vec{v}$ is the eigenvector of A corresponding to r . There are three possibilities of eigenvalues of A

- n distinct real eigenvalues
- complex conjugate eigenvalues
- eigenvalues with multiplicity k

1. When all eigenvalues $r_1, \dots, r_n$ are distinct with n different eigenvectors $v_1, \dots, v_n$ then the solution is given by

$$\vec{y} = C_1 e^{r_1 t} \vec{v}_1 + C_2 e^{r_2 t} \vec{v}_2 + \dots + C_n e^{r_n t} \vec{v}_n$$

where $C_1, C_2, \dots, C_n$ are arbitrary constants.

2. When some eigenvalues are complex then in order to make the solution real, we take a linear combination: if $r = a + bi$ has an eigenvector $\vec{v} = \vec{w}_1 + i\vec{w}_2$ then to obtain real-valued solutions to the system, replace the complex-valued solutions $e^{rx} \vec{v}$ with real-valued solution $e^{ax}(\vec{w}_1 \cos(bx) - \vec{w}_2 \sin(bx))$ and for $r = a - bi$ replace the solution $e^{-rx} \vec{v}$ with $e^{ax}(\vec{w}_1 \sin(bx) + \vec{w}_2 \cos(bx))$

3. If some eigenvalues are repeated. Then we get fewer than n linearly independent eigenvectors, we miss some of the solutions and need to construct the missing ones. We do this via generalized eigenvectors, vectors which are not eigenvectors but are close enough that we can use to write down the remaining solutions. For a eigenvalue r with eigenvector $\vec{w}$ we obtain $\vec{w}_2, \dots, \vec{w}_k$ using

$$(A - rI) \cdot \vec{w}_2 = \vec{w}$$

$$(A - rI) \cdot \vec{w}_3 = \vec{w}_2$$

$$\vdots$$

$$(A - rI) \cdot \vec{w}_k = \vec{w}_{k-1}$$

Then the solutions to the system for the eigenspace are $e^{rt}[\vec{w}]$, $e^{rt}[t\vec{w} + \vec{w}_2]$, $e^{rt}[\frac{t^2}{2}\vec{w} + t\vec{w}_2 + \vec{w}_3]$, ..., $e^{rt}[\frac{t^{k-1}}{(k-1)!}\vec{w} + \frac{t^{k-2}}{(k-2)!}\vec{w}_2 + \dots + t\vec{w}_{k-1} + \vec{w}_k]$

So, If $\vec{y}_1, \dots, \vec{y}_n$ are n solution of obtained from three categories of A , then general solution to the system $\vec{y}' = A \cdot \vec{y}$

$$\vec{y} = C_1 \vec{y}_1 + C_2 \vec{y}_2 + \dots + C_n \vec{y}_n$$

system_of_odes_nonlinear_2eq_order1_type1

`sympy.solvers.ode._nonlinear_2eq_order1_type1(x, y, t, eq)`

Equations:

$$x' = x^n F(x, y)$$

$$y' = g(y)F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if $n \neq 1$

$$\varphi = [C_1 + (1 - n) \int \frac{1}{g(y)} dy]^{\frac{1}{1-n}}$$

if $n = 1$

$$\varphi = C_1 e^{\int \frac{1}{g(y)} dy}$$

where C_1 and C_2 are arbitrary constants.

system_of_odes_nonlinear_2eq_order1_type2`sympy.solvers.ode._nonlinear_2eq_order1_type2(x, y, t, eq)`

Equations:

$$x' = e^{\lambda x} F(x, y)$$

$$y' = g(y) F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y) F(\varphi(y), y)} dy = t + C_2$$

where

if $\lambda \neq 0$

$$\varphi = -\frac{1}{\lambda} \log(C_1 - \lambda \int \frac{1}{g(y)} dy)$$

if $\lambda = 0$

$$\varphi = C_1 + \int \frac{1}{g(y)} dy$$

where C_1 and C_2 are arbitrary constants.**system_of_odes_nonlinear_2eq_order1_type3**`sympy.solvers.ode._nonlinear_2eq_order1_type3(x, y, t, eq)`

Autonomous system of general form

$$x' = F(x, y)$$

$$y' = G(x, y)$$

Assuming $y = y(x, C_1)$ where C_1 is an arbitrary constant is the general solution of the first-order equation

$$F(x, y) y'_x = G(x, y)$$

Then the general solution of the original system of equations has the form

$$\int \frac{1}{F(x, y(x, C_1))} dx = t + C_1$$

system_of_odes_nonlinear_2eq_order1_type4`sympy.solvers.ode._nonlinear_2eq_order1_type4(x, y, t, eq)`

Equation:

$$x' = f_1(x) g_1(y) \phi(x, y, t)$$

$$y' = f_2(x) g_2(y) \phi(x, y, t)$$

First integral:

$$\int \frac{f_2(x)}{f_1(x)} dx - \int \frac{g_1(y)}{g_2(y)} dy = C$$

where C is an arbitrary constant.

On solving the first integral for x (resp., y) and on substituting the resulting expression into either equation of the original solution, one arrives at a first-order equation for determining y (resp., x).

system_of_odes_nonlinear_2eq_order1_type5

`sympy.solvers.ode._nonlinear_2eq_order1_type5(func, t, eq)`
Clairaut system of ODEs

$$x = tx' + F(x', y')$$

$$y = ty' + G(x', y')$$

The following are solutions of the system

(i) straight lines:

$$x = C_1 t + F(C_1, C_2), y = C_2 t + G(C_1, C_2)$$

where C_1 and C_2 are arbitrary constants;

(ii) envelopes of the above lines;

(iii) continuously differentiable lines made up from segments of the lines (i) and (ii).

system_of_odes_nonlinear_3eq_order1_type1

`sympy.solvers.ode._nonlinear_3eq_order1_type1(x, y, z, t, eq)`
Equations:

$$ax' = (b - c)yz, \quad by' = (c - a)zx, \quad cz' = (a - b)xy$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrive at a separable first-order equation on x . Similarly doing that for other two equations, we will arrive at first order equation on y and z too.

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0401.pdf>

system_of_odes_nonlinear_3eq_order1_type2

`sympy.solvers.ode._nonlinear_3eq_order1_type2(x, y, z, t, eq)`

Equations:

$$ax' = (b - c)yzf(x, y, z, t)$$

$$by' = (c - a)zxf(x, y, z, t)$$

$$cz' = (a - b)xyf(x, y, z, t)$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrive at a first-order differential equation on x . Similarly doing that for other two equations we will arrive at first order equation on y and z .

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0402.pdf>

system_of_odes_nonlinear_3eq_order1_type3

`sympy.solvers.ode._nonlinear_3eq_order1_type3(x, y, z, t, eq)`

Equations:

$$x' = cF_2 - bF_3, \quad y' = aF_3 - cF_1, \quad z' = bF_1 - aF_2$$

where $F_n = F_n(x, y, z, t)$.

1. First Integral:

$$ax + by + cz = C_1,$$

where C is an arbitrary constant.

2. If we assume function F_n to be independent of t , i.e., $F_n = F_n(x, y, z)$ Then, on eliminating t and z from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{aF_3(x, y, z) - cF_1(x, y, z)}{cF_2(x, y, z) - bF_3(x, y, z)}$$

where $z = \frac{1}{c}(C_1 - ax - by)$

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0404.pdf>

system_of_odes_nonlinear_3eq_order1_type4

`sympy.solvers.ode._nonlinear_3eq_order1_type4(x, y, z, t, eq)`

Equations:

$$x' = czF_2 - byF_3, \quad y' = axF_3 - czF_1, \quad z' = byF_1 - axF_2$$

where $F_n = F_n(x, y, z, t)$

1. First integral:

$$ax^2 + by^2 + cz^2 = C_1$$

where C is an arbitrary constant.

2. Assuming the function F_n is independent of t : $F_n = F_n(x, y, z)$. Then on eliminating t and z from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{axF_3(x, y, z) - czF_1(x, y, z)}{czF_2(x, y, z) - byF_3(x, y, z)}$$

where $z = \pm \sqrt{\frac{1}{c}(C_1 - ax^2 - by^2)}$

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0405.pdf>

system_of_odes_nonlinear_3eq_order1_type5

`sympy.solvers.ode._nonlinear_3eq_order1_type5(x, y, t, eq)`

$$x' = x(cF_2 - bF_3), \quad y' = y(aF_3 - cF_1), \quad z' = z(bF_1 - aF_2)$$

where $F_n = F_n(x, y, z, t)$ and are arbitrary functions.

First Integral:

$$|x|^a |y|^b |z|^c = C_1$$

where C is an arbitrary constant. If the function F_n is independent of t , then, by eliminating t and z from the first two equations of the system, one arrives at a first-order equation.

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0406.pdf>

5.29.5 Information on the ode module

This module contains `dsolve()` (page 1230) and different helper functions that it uses.

`dsolve()` (page 1230) solves ordinary differential equations. See the docstring on the various functions for their uses. Note that partial differential equations support is in `pde.py`. Note that hint functions have docstrings describing their various methods, but they are intended

for internal use. Use `dsolve(ode, func, hint=hint)` to solve an ODE using a specific hint. See also the docstring on `dsolve()` (page 1230).

Functions in this module

These are the user functions in this module:

- `dsolve()` (page 1230) - Solves ODEs.
- `classify_ode()` (page 1233) - Classifies ODEs into possible hints for `dsolve()` (page 1230).
- `checkodesol()` (page 1235) - Checks if an equation is the solution to an ODE.
- `homogeneous_order()` (page 1235) - Returns the homogeneous order of an expression.
- `infinitesimals()` (page 1236) - Returns the infinitesimals of the Lie group of point transformations of an ODE, such that it is invariant.
- `ode_checkinfsol()` - Checks if the given infinitesimals are the actual infinitesimals of a first order ODE.

These are the non-solver helper functions that are for internal use. The user should use the various options to `dsolve()` (page 1230) to obtain the functionality provided by these functions:

- `odesimp()` (page 1238) - Does all forms of ODE simplification.
- `ode_sol_simplicity()` (page 1241) - A key function for comparing solutions by simplicity.
- `constantsimp()` (page 1240) - Simplifies arbitrary constants.
- `constant_renumber()` (page 1239) - Renumber arbitrary constants.
- `_handle_Integral()` - Evaluate unevaluated Integrals.

See also the docstrings of these functions.

Currently implemented solver methods

The following methods are implemented for solving ordinary differential equations. See the docstrings of the various hint functions for more information on each (run `help(ode)`):

- 1st order separable differential equations.
- 1st order differential equations whose coefficients or dx and dy are functions homogeneous of the same order.
- 1st order exact differential equations.
- 1st order linear differential equations.
- 1st order Bernoulli differential equations.
- Power series solutions for first order differential equations.
- Lie Group method of solving first order differential equations.
- 2nd order Liouville differential equations.
- Power series solutions for second order differential equations at ordinary and regular singular points.
- n th order linear homogeneous differential equation with constant coefficients.
- n th order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.

- n th order linear inhomogeneous differential equation with constant coefficients using the method of variation of parameters.

Philosophy behind this module

This module is designed to make it easy to add new ODE solving methods without having to mess with the solving code for other methods. The idea is that there is a `classify_ode()` (page 1233) function, which takes in an ODE and tells you what hints, if any, will solve the ODE. It does this without attempting to solve the ODE, so it is fast. Each solving method is a hint, and it has its own function, named `ode_<hint>`. That function takes in the ODE and any match expression gathered by `classify_ode()` (page 1233) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 690) class. `dsolve()` (page 1230), which is the user wrapper function around all of this, will then call `odesimp()` (page 1238) on the result, which, among other things, will attempt to solve the equation for the dependent variable (the function we are solving for), simplify the arbitrary constants in the expression, and evaluate any integrals, if the hint allows it.

How to add new solution methods

If you have an ODE that you want `dsolve()` (page 1230) to be able to solve, try to avoid adding special case code here. Instead, try finding a general method that will solve your ODE, as well as others. This way, the `ode` (page 1283) module will become more robust, and unhindered by special case hacks. WolframAlpha and Maple's `DETools[odeadvisor]` function are two resources you can use to classify a specific ODE. It is also better for a method to work with an n th order ODE instead of only with specific orders, if possible.

To add a new method, there are a few things that you need to do. First, you need a hint name for your method. Try to name your hint so that it is unambiguous with all other methods, including ones that may not be implemented yet. If your method uses integrals, also include a `hint_Integral` hint. If there is more than one way to solve ODEs with your method, include a hint for each one, as well as a `<hint>_best` hint. Your `ode_<hint>_best()` function should choose the best using `min` with `ode_sol_simplicity` as the key argument. See `ode_1st_homogeneous_coeff_best()` (page 1243), for example. The function that uses your method will be called `ode_<hint>()`, so the hint must only use characters that are allowed in a Python function name (alphanumeric characters and the underscore `'_'` character). Include a function for every hint, except for `_Integral` hints (`dsolve()` (page 1230) takes care of those automatically). Hint names should be all lowercase, unless a word is commonly capitalized (such as `Integral` or `Bernoulli`). If you have a hint that you do not want to run with `all_Integral` that doesn't have an `_Integral` counterpart (such as a `best` hint that would defeat the purpose of `all_Integral`), you will need to remove it manually in the `dsolve()` (page 1230) code. See also the `classify_ode()` (page 1233) docstring for guidelines on writing a hint name.

Determine in general how the solutions returned by your method compare with other methods that can potentially solve the same ODEs. Then, put your hints in the `allhints` (page 1238) tuple in the order that they should be called. The ordering of this tuple determines which hints are default. Note that exceptions are ok, because it is easy for the user to choose individual hints with `dsolve()` (page 1230). In general, `_Integral` variants should go at the end of the list, and `_best` variants should go before the various hints they apply to. For example, the `undetermined_coefficients` hint comes before the `variation_of_parameters` hint because, even though variation of parameters is more general than undetermined coefficients, undetermined coefficients generally returns cleaner results for the ODEs that it can solve than variation of parameters does, and it does not require integration, so it is much faster.

Next, you need to have a match expression or a function that matches the type of the ODE, which you should put in `classify_ode()` (page 1233) (if the match function is more than just a few lines, like `_undetermined_coefficients_match()`, it should go outside of `classify_ode()` (page 1233)). It should match the ODE without solving for it as much as possible,

so that `classify_ode()` (page 1233) remains fast and is not hindered by bugs in solving code. Be sure to consider corner cases. For example, if your solution method involves dividing by something, make sure you exclude the case where that division will be 0.

In most cases, the matching of the ODE will also give you the various parts that you need to solve it. You should put that in a dictionary (`.match()` will do this for you), and add that as `matching_hints['hint'] = matchdict` in the relevant part of `classify_ode()` (page 1233). `classify_ode()` (page 1233) will then send this to `dsolve()` (page 1230), which will send it to your function as the `match` argument. Your function should be named `ode_<hint>(eq, func, order, match)`. If you need to send more information, put it in the `match` dictionary. For example, if you had to substitute in a dummy variable in `classify_ode()` (page 1233) to match the ODE, you will need to pass it to your function using the `match` dict to access it. You can access the independent variable using `func.args[0]`, and the dependent variable (the function you are trying to solve for) as `func.func`. If, while trying to solve the ODE, you find that you cannot, raise `NotImplementedError`. `dsolve()` (page 1230) will catch this error with the `all` meta-hint, rather than causing the whole routine to fail.

Add a docstring to your function that describes the method employed. Like with anything else in SymPy, you will need to add a doctest to the docstring, in addition to real tests in `test_ode.py`. Try to maintain consistency with the other hint functions' docstrings. Add your method to the list at the top of this docstring. Also, add your method to `ode.rst` in the `docs/src` directory, so that the Sphinx docs will pull its docstring into the main SymPy documentation. Be sure to make the Sphinx documentation by running `make html` from within the doc directory to verify that the docstring formats correctly.

If your solution method involves integrating, use `Integral()` instead of `integrate()` (page 124). This allows the user to bypass hard/slow integration by using the `_Integral` variant of your hint. In most cases, calling `sympy.core.basic.Basic.doit()` (page 100) will integrate your solution. If this is not the case, you will need to write special code in `_handle_Integral()`. Arbitrary constants should be symbols named `C1`, `C2`, and so on. All solution methods should return an equality instance. If you need an arbitrary number of arbitrary constants, you can use `constants = numbered_symbols(prefix='C', cls=Symbol, start=1)`. If it is possible to solve for the dependent function in a general way, do so. Otherwise, do as best as you can, but do not call `solve` in your `ode_<hint>()` function. `odesimp()` (page 1238) will attempt to solve the solution for you, so you do not need to do that. Lastly, if your ODE has a common simplification that can be applied to your solutions, you can add a special case in `odesimp()` (page 1238) for it. For example, solutions returned from the `1st_homogeneous_coeff` hints often have many `log()` terms, so `odesimp()` (page 1238) calls `logcombine()` (page 1161) on them (it also helps to write the arbitrary constant as `log(C1)` instead of `C1` in this case). Also consider common ways that you can rearrange your solution to have `constantsimp()` (page 1240) take better advantage of it. It is better to put simplification in `odesimp()` (page 1238) than in your method, because it can then be turned off with the `simplify` flag in `dsolve()` (page 1230). If you have any extraneous simplification in your function, be sure to only run it using `if match.get('simplify', True):`, especially if it can be slow or if it can reduce the domain of the solution.

Finally, as with every contribution to SymPy, your method will need to be tested. Add a test for each method in `test_ode.py`. Follow the conventions there, i.e., test the solver using `dsolve(eq, f(x), hint=your_hint)`, and also test the solution using `checkodesol()` (page 1235) (you can put these in a separate tests and skip/XFAIL if it runs too slow/doesn't work). Be sure to call your hint specifically in `dsolve()` (page 1230), that way the test won't be broken simply by the introduction of another matching hint. If your method works for higher order (>1) ODEs, you will need to run `sol = constant_renumber(sol, 'C', 1, order)` for each solution, where `order` is the order of the ODE. This is because `constant_renumber` renumbers the arbitrary constants by printing order, which is platform dependent. Try to

test every corner case of your solver, including a range of orders if it is a n th order solver, but if your solver is slow, such as if it involves hard integration, try to keep the test run time down.

Feel free to refactor existing hints to avoid duplicating code or creating inconsistencies. If you can show that your method exactly duplicates an existing method, including in the simplicity and speed of obtaining the solutions, then you can remove the old, less general method. The existing code is tested extensively in `test_ode.py`, so if anything is broken, one of those tests will surely fail.

5.30 PDE

5.30.1 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. They are intended for user use.

`pde_separate()`

`sympy.solvers.pde.pde_separate(eq, fun, sep, strategy='mul')`

Separate variables in partial differential equation either by additive or multiplicative separation approach. It tries to rewrite an equation so that one of the specified variables occurs on a different side of the equation than the others.

Parameters

- **eq** – Partial differential equation
- **fun** – Original function $F(x, y, z)$
- **sep** – List of separated functions $[X(x), u(y, z)]$
- **strategy** – Separation strategy. You can choose between additive separation ('add') and multiplicative separation ('mul') which is default.

See also:

[pde_separate_add](#) (page 1288), [pde_separate_mul](#) (page 1288)

Examples

```
>>> from sympy import E, Eq, Function, pde_separate, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

```
>>> eq = Eq(D(u(x, t), x, 2), D(u(x, t), t, 2))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='mul')
[Derivative(X(x), (x, 2))/X(x), Derivative(T(t), (t, 2))/T(t)]
```

pde_separate_add()

`sympy.solvers.pde.pde_separate_add(eq, fun, sep)`

Helper function for searching additive separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w , we look for the product of two functions depending on different arguments:

$$w(x, y, z) = X(x) + y(y, z)$$

Examples

```
>>> from sympy import E, Eq, Function, pde_separate_add, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

pde_separate_mul()

`sympy.solvers.pde.pde_separate_mul(eq, fun, sep)`

Helper function for searching multiplicative separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w , we look for the product of two functions depending on different arguments:

$$w(x, y, z) = X(x) * u(y, z)$$

Examples

```
>>> from sympy import Function, Eq, pde_separate_mul, Derivative as D
>>> from sympy.abc import x, y
>>> u, X, Y = map(Function, 'uXY')
```

```
>>> eq = Eq(D(u(x, y), x, 2), D(u(x, y), y, 2))
>>> pde_separate_mul(eq, u(x, y), [X(x), Y(y)])
[Derivative(X(x), (x, 2))/X(x), Derivative(Y(y), (y, 2))/Y(y)]
```

pdsolve()

`sympy.solvers.pde.pdsolve(eq, func=None, hint='default', dict=False, solve-
fun=None, **kwargs)`

Solves any (supported) kind of partial differential equation.

Usage

`pdsolve(eq, f(x,y), hint)` -> Solve partial differential equation `eq` for function `f(x,y)`, using method `hint`.

Details

eq can be any supported partial differential equation (see the `pde` docstring for supported methods). This can either be an Equality, or an expression, which is assumed to be equal to 0.

f(x,y) is a function of two variables whose derivatives in that variable make up the partial differential equation. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it couldn't be detected).

hint is the solving method that you want `pdsolve` to use. Use `classify_pde(eq, f(x,y))` to get all of the possible hints for a PDE. The default hint, 'default', will use whatever hint is returned first by `classify_pde()`. See Hints below for more options that you can use for hint.

solvefun is the convention used for arbitrary functions returned by the PDE solver. If not set by the user, it is set by default to be `F`.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to `pdsolve()`:

"default": This uses whatever hint is returned first by `classify_pde()`. This is the default argument to `pdsolve()`.

"all": To make `pdsolve` apply all relevant classification hints, use `pdsolve(PDE, func, hint="all")`. This will return a dictionary of hint:solution terms. If a hint causes `pdsolve` to raise the `NotImplementedError`, value of that hint's key will be the exception object raised. The dictionary will also include some special keys:

- **order**: The order of the PDE. See also `ode_order()` in `deutils.py`
- **default**: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by `classify_pde()`.

"all_Integral": This is the same as "all", except if a hint also has a corresponding "_Integral" hint, it only returns the "_Integral" hint. This is useful if "all" causes `pdsolve()` to hang because of a difficult or impossible integral. This meta-hint will also be much faster than "all", because `integrate()` is an expensive routine.

See also the `classify_pde()` docstring for more info on hints, and the `pde` docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x, y # x and y are the independent variables
>>> f = Function("f")(x, y) # f is a function of x and y
>>> # fx will be the partial derivative of f with respect to x
>>> fx = Derivative(f, x)
>>> # fy will be the partial derivative of f with respect to y
>>> fy = Derivative(f, y)
```

- See `test_pde.py` for many tests, which serves also as a set of examples for how to use `pdsolve()`.

- `pdsolve` always returns an Equality class (except for the case when the hint is “all” or “all_Integral”). Note that it is not possible to get an explicit solution for $f(x, y)$ as in the case of ODE’s
- Do `help(pde.pde_hintname)` to get help more information on a specific hint

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, diff, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)))
>>> pdsolve(eq)
Eq(f(x, y), F(3*x - 2*y)*exp(-2*x/13 - 3*y/13))
```

`classify_pde()`

`sympy.solvers.pde.classify_pde(eq, func=None, dict=False, **kwargs)`

Returns a tuple of possible `pdsolve()` classifications for a PDE.

The tuple is ordered so that first item is the classification that `pdsolve()` uses to solve the PDE by default. In general, classifications near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make `pdsolve` use a different classification, use `pdsolve(PDE, func, hint=<classification>)`. See also the `pdsolve()` docstring for different meta-hints you can use.

If `dict` is true, `classify_pde()` will return a dictionary of `hint:match` expression terms. This is intended for internal use by `pdsolve()`. Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by doing `help(pde.pde_hintname)`, where `hintname` is the name of the hint without “_Integral”.

See `sympy.pde.allhints` or the `sympy.pde` docstring for a list of all supported hints that can be returned from `classify_pde`.

Examples

```
>>> from sympy.solvers.pde import classify_pde
>>> from sympy import Function, diff, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)))
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)
```


checkpdesol()

`sympy.solvers.pde.checkpdesol(pde, sol, func=None, solve_for_func=True)`

Checks if the given solution satisfies the partial differential equation.

`pde` is the partial differential equation which can be given in the form of an equation or an expression. `sol` is the solution for which the `pde` is to be checked. This can also be given in an equation or an expression form. If the function is not provided, the helper function `_preprocess` from `deutils` is used to identify the function.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

The following methods are currently being implemented to check if the solution satisfies the PDE:

1. Directly substitute the solution in the PDE and check. If the solution hasn't been solved for `f`, then it will solve for `f` provided `solve_for_func` hasn't been set to `False`.

If the solution satisfies the PDE, then a tuple `(True, 0)` is returned. Otherwise a tuple `(False, expr)` where `expr` is the value obtained after substituting the solution in the PDE. However if a known solution returns `False`, it may be due to the inability of `doit()` to simplify it to zero.

Examples

```
>>> from sympy import Function, symbols, diff
>>> from sympy.solvers.pde import checkpdesol, pdsolve
>>> x, y = symbols('x y')
>>> f = Function('f')
>>> eq = 2*f(x,y) + 3*f(x,y).diff(x) + 4*f(x,y).diff(y)
>>> sol = pdsolve(eq)
>>> assert checkpdesol(eq, sol)[0]
>>> eq = x*f(x,y) + f(x,y).diff(x)
>>> checkpdesol(eq, sol)
(False, (x*(F(4*x - 3*y) - 6*(F(4*x - 3*y)/25 + 4*Subs(Derivative(F(_xi_1), _xi_1),
↪(_xi_1, (4*x - 3*y))))*exp(-6*x/25 - 8*y/25))
```

5.30.2 Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

pde_1st_linear_constant_coeff_homogeneous

`sympy.solvers.pde.pde_1st_linear_constant_coeff_homogeneous(eq, func, order, match, solvefun)`

Solves a first order linear homogeneous partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{df(x,y)}{dx} + b \frac{df(x,y)}{dy} + cf(x,y) = 0$$

where a , b and c are constants.

The general solution is of the form:

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform)
      d      d
a*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y)
   dx          dy

>>> pprint(pdsolve(genform))
      -c*(a*x + b*y)
      -----
           2    2
          a  + b

f(x, y) = F(-a*y + b*x)*e
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

Examples

```
>>> from sympy.solvers.pde import (
... pde_1st_linear_constant_coeff_homogeneous)
>>> from sympy import pdsolve
>>> from sympy import Function, diff, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y))
Eq(f(x, y), F(x - y)*exp(-x/2 - y/2))
>>> pprint(pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y)))
      x    y
      - - - -
      2    2

f(x, y) = F(x - y)*e
```

pde_1st_linear_constant_coeff

`sympy.solvers.pde.pde_1st_linear_constant_coeff`(eq, func, order, match, solve-
fun)

Solves a first order linear partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{df(x,y)}{dx} + b \frac{df(x,y)}{dy} + cf(x,y) = G(x,y)$$

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, diff, pprint, exp
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), (F(4*x + 2*y) + exp(x/2 + 4*y)/15)*exp(x/2 - y))
```

pde_1st_linear_variable_coeff

`sympy.solvers.pde.pde_1st_linear_variable_coeff(eq, func, order, match, solve-
fun)`

Solves a first order linear partial differential equation with variable coefficients. The general form of this partial differential equation is

$$a(x,y)\frac{df(x,y)}{dx} + b(x,y)\frac{df(x,y)}{dy} + c(x,y)f(x,y) - G(x,y)$$

where $a(x,y)$, $b(x,y)$, $c(x,y)$ and $G(x,y)$ are arbitrary functions in x and y . This PDE is converted into an ODE by making the following transformation.

1] ξ as x

2] η as the constant in the solution to the differential equation $\frac{dy}{dx} = -\frac{b}{a}$

Making the following substitutions reduces it to the linear ODE

$$a(\xi,\eta)\frac{du}{d\xi} + c(\xi,\eta)u - d(\xi,\eta) = 0$$

which can be solved using `dsolve`.

The general form of this PDE is:

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy.abc import x, y
>>> from sympy import Function, pprint
>>> a, b, c, G, f = [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x,y)
>>> pprint(genform)
-G(x, y) + a(x, y)*f(x, y) + b(x, y)*--(f(x, y)) + c(x, y)*--(f(x, y))
                                dx                                dy
```

References

- Viktor Grigoryan, "Partial Differential Equations" Math 124A - Fall 2010, pp.7

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, diff, pprint, exp
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), F(x*y)*exp(y**2/2) + 1)
```

5.30.3 Information on the pde module

This module contains `pdsolve()` and different helper functions that it uses. It is heavily inspired by the `ode` module and hence the basic infrastructure remains the same.

Functions in this module

These are the user functions in this module:

- `pdsolve()` - Solves PDE's
- `classify_pde()` - Classifies PDEs into possible hints for `dsolve()`.
- **`pde_separate()` - Separate variables in partial differential equation either by additive or multiplicative separation approach.**

These are the helper functions in this module:

- `pde_separate_add()` - Helper function for searching additive separable solutions.
- **`pde_separate_mul()` - Helper function for searching multiplicative separable solutions.**

Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various `pde_hint()` functions for more information on each (run `help(pde)`):

- 1st order linear homogeneous partial differential equations with constant coefficients.
- 1st order linear general partial differential equations with constant coefficients.
- 1st order linear partial differential equations with variable coefficients.

5.31 Solvers

The solvers module in SymPy implements methods for solving equations.

Note: It is recommended to use `solveset()` (page 1342) to solve univariate equations, `sympy.solvers.solveset.linsolve()` (page 1354) to solve system of linear equations instead of `solve()` and `sympy.solvers.solveset.nonlinsolve()` (page 1357) to solve system of non linear equations since sooner or later the `solveset` will take over `solve` either internally or externally.

5.31.1 Algebraic equations

Use `solve()` to solve algebraic equations. We suppose all equations are equaled to 0, so solving `x**2 == 1` translates into the following code:

```
>>> from sympy.solvers import solve
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> solve(x**2 - 1, x)
[-1, 1]
```

The first argument for `solve()` is an equation (equaled to zero) and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solvers.solve(f, *symbols, **flags)`
Algebraically solves equations and systems of equations.

Currently supported are:

- polynomial,
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- systems containing relational expressions.

Input is formed as:

- **f**
 - a single Expr or Poly that must be zero,
 - an Equality
 - a Relational expression or boolean
 - iterable of one or more of the above
- **symbols (object(s) to solve for) specified as**
 - none given (other non-numeric objects will be used)
 - single symbol
 - denested list of symbols e.g. `solve(f, x, y)`
 - ordered iterable of symbols e.g. `solve(f, [x, y])`
- **flags**
 - 'dict'=True (default is False)** return list (perhaps empty) of solution mappings
 - 'set'=True (default is False)** return list of symbols and set of tuple(s) of solution(s)
 - 'exclude=[] (default)'** don't try to solve for any of the free symbols in exclude; if expressions are given, the free symbols in them will be extracted automatically.
 - 'check=True (default)'** If False, don't do any testing of solutions. This can be useful if one wants to include solutions that make any denominator zero.
 - 'numerical=True (default)'** do a fast numerical check if f has only one symbol.

'minimal=True (default is False)' a very fast, minimal testing.

'warn=True (default is False)' show a warning if `checksol()` could not conclude.

'simplify=True (default)' simplify all but polynomials of order 3 or greater before returning them and (if `check` is not `False`) use the general `simplify` function on the solutions and the expression obtained when they are substituted into the function which should be zero

'force=True (default is False)' make positive all symbols without assumptions regarding sign.

'rational=True (default)' recast Floats as Rational; if this option is not used, the system containing floats may fail to solve because of issues with polys. If `rational=None`, Floats will be recast as rationals but the answer will be recast as Floats. If the flag is `False` then nothing will be done to the Floats.

'manual=True (default is False)' do not use the polys/matrix method to solve a system of equations, solve them one at a time as you might "manually"

'implicit=True (default is False)' allows `solve` to return a solution for a pattern in terms of other functions that contain that pattern; this is only needed if the pattern is inside of some invertible function like `cos`, `exp`, ...

'particular=True (default is False)' instructs `solve` to try to find a particular solution to a linear system with as many zeros as possible; this is very expensive

'quick=True (default is False)' when using `particular=True`, use a fast heuristic instead to find a solution with many zeros (instead of using the very slow method guaranteed to find the largest number of zeros possible)

'cubics=True (default)' return explicit solutions when cubic expressions are encountered

'quartics=True (default)' return explicit solutions when quartic expressions are encountered

'quintics=True (default)' return explicit solutions (if possible) when quintic expressions are encountered

Notes

`solve()` with `check=True` (default) will run through the symbol tags to eliminate unwanted solutions. If no assumptions are included all possible solutions will be returned.

```
>>> from sympy import Symbol, solve
>>> x = Symbol("x")
>>> solve(x**2 - 1)
[-1, 1]
```

By using the positive tag only one solution will be returned:

```
>>> pos = Symbol("pos", positive=True)
>>> solve(pos**2 - 1)
[1]
```

Assumptions aren't checked when `solve()` input involves relationals or bools.

When the solutions are checked, those that make any denominator zero are automatically excluded. If you do not want to exclude such solutions then use the `check=False` option:

```
>>> from sympy import sin, limit
>>> solve(sin(x)/x) # 0 is excluded
[pi]
```

If `check=False` then a solution to the numerator being zero is found: $x = 0$. In this case, this is a spurious solution since $\sin(x)/x$ has the well known limit (without discontinuity) of 1 at $x = 0$:

```
>>> solve(sin(x)/x, check=False)
[0, pi]
```

In the following case, however, the limit exists and is equal to the value of $x = 0$ that is excluded when `check=True`:

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve(eq, x)
[]
>>> solve(eq, x, check=False)
[0]
>>> limit(eq, x, 0, '-')
0
>>> limit(eq, x, 0, '+')
0
```

Examples

The output varies according to the input and can be seen by example:

```
>>> from sympy import solve, Poly, Eq, Function, exp
>>> from sympy.abc import x, y, z, a, b
>>> f = Function('f')
```

- boolean or univariate Relational

```
>>> solve(x < 3)
(-oo < x) & (x < 3)
```

- to always get a list of solution mappings, use flag `dict=True`

```
>>> solve(x - 3, dict=True)
[{x: 3}]
>>> sol = solve([x - 3, y - 1], dict=True)
>>> sol
[{x: 3, y: 1}]
>>> sol[0][x]
3
>>> sol[0][y]
1
```

- to get a list of symbols and set of solution(s) use flag `set=True`

```
>>> solve([x**2 - 3, y - 1], set=True)
([x, y], {(-sqrt(3), 1), (sqrt(3), 1)})
```


- single expression and single symbol that is in the expression

```
>>> solve(x - y, x)
[y]
>>> solve(x - 3, x)
[3]
>>> solve(Eq(x, 3), x)
[3]
>>> solve(Poly(x - 3), x)
[3]
>>> solve(x**2 - y**2, x, set=True)
([x], {(-y,), (y,)})
>>> solve(x**4 - 1, x, set=True)
([x], {(-1,), (1,), (-I,), (I,)})
```

- single expression with no symbol that is in the expression

```
>>> solve(3, x)
[]
>>> solve(x - 3, y)
[]
```

- single expression with no symbol given

In this case, all free symbols will be selected as potential symbols to solve for. If the equation is univariate then a list of solutions is returned; otherwise – as is the case when symbols are given as an iterable of length > 1 – a list of mappings will be returned.

```
>>> solve(x - 3)
[3]
>>> solve(x**2 - y**2)
[{x: -y}, {x: y}]
>>> solve(z**2*x**2 - z**2*y**2)
[{x: -y}, {x: y}, {z: 0}]
>>> solve(z**2*x - z**2*y**2)
[{x: y**2}, {z: 0}]
```

- when an object other than a Symbol is given as a symbol, it is isolated algebraically and an implicit solution may be obtained. This is mostly provided as a convenience to save one from replacing the object with a Symbol and solving for that Symbol. It will only work if the specified object can be replaced with a Symbol using the subs method.

```
>>> solve(f(x) - x, f(x))
[x]
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))
[x + f(x)]
>>> solve(f(x).diff(x) - f(x) - x, f(x))
[-x + Derivative(f(x), x)]
>>> solve(x + exp(x)**2, exp(x), set=True)
([exp(x)], {(-sqrt(-x),), (sqrt(-x),)})
```

```
>>> from sympy import Indexed, IndexedBase, Tuple, sqrt
>>> A = IndexedBase('A')
>>> eqs = Tuple(A[1] + A[2] - 3, A[1] - A[2] + 1)
>>> solve(eqs, eqs.atoms(Indexed))
{A[1]: 1, A[2]: 2}
```

- To solve for a symbol implicitly, use 'implicit=True':

```
>>> solve(x + exp(x), x)
[-LambertW(1)]
>>> solve(x + exp(x), x, implicit=True)
[-exp(x)]
```

- It is possible to solve for anything that can be targeted with subs:

```
>>> solve(x + 2 + sqrt(3), x + 2)
[-sqrt(3)]
>>> solve((x + 2 + sqrt(3), x + 4 + y), y, x + 2)
{y: -2 + sqrt(3), x + 2: -sqrt(3)}
```

- Nothing heroic is done in this implicit solving so you may end up with a symbol still in the solution:

```
>>> eqs = (x*y + 3*y + sqrt(3), x + 4 + y)
>>> solve(eqs, y, x + 2)
{y: -sqrt(3)/(x + 3), x + 2: (-2*x - 6 + sqrt(3))/(x + 3)}
>>> solve(eqs, y*x, x)
{x: -y - 4, x*y: -3*y - sqrt(3)}
```

- if you attempt to solve for a number remember that the number you have obtained does not necessarily mean that the value is equivalent to the expression obtained:

```
>>> solve(sqrt(2) - 1, 1)
[sqrt(2)]
>>> solve(x - y + 1, 1) # /\ -1 is targeted, too
[x/(y - 1)]
>>> [_subs(z, -1) for _ in solve((x - y + 1).subs(-1, z), 1)]
[-x + y]
```

- To solve for a function within a derivative, use dsolve.

- single expression and more than 1 symbol

- when there is a linear solution

```
>>> solve(x - y**2, x, y)
[(y**2, y)]
>>> solve(x**2 - y, x, y)
[(x, x**2)]
>>> solve(x**2 - y, x, y, dict=True)
[{y: x**2}]
```

- when undetermined coefficients are identified

- * that are linear

```
>>> solve((a + b)*x - b + 2, a, b)
{a: -2, b: 2}
```

- * that are nonlinear

```
>>> solve((a + b)*x - b**2 + 2, a, b, set=True)
([a, b], {(-sqrt(2), sqrt(2)), (sqrt(2), -sqrt(2))})
```

- if there is no linear solution then the first successful attempt for a nonlinear solution will be returned

```
>>> solve(x**2 - y**2, x, y, dict=True)
[{x: -y}, {x: y}]
>>> solve(x**2 - y**2/exp(x), x, y, dict=True)
[{x: 2*LambertW(y/2)}]
>>> solve(x**2 - y**2/exp(x), y, x)
[(-x*sqrt(exp(x)), x), (x*sqrt(exp(x)), x)]
```

- iterable of one or more of the above
 - involving relationals or bools

```
>>> solve([x < 3, x - 2])
Eq(x, 2)
>>> solve([x > 3, x - 2])
False
```

- when the system is linear
 - * with a solution

```
>>> solve([x - 3], x)
{x: 3}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y, z)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - z), z, x, y)
{x: -5*y + 2, z: 21*y - 6}
```

- * without a solution

```
>>> solve([x + 3, x - 3])
[]
```

- when the system is not linear

```
>>> solve([x**2 + y - 2, y**2 - 4], x, y, set=True)
([x, y], {(-2, -2), (0, 2), (2, -2)})
```

- if no symbols are given, all free symbols will be selected and a list of mappings returned

```
>>> solve([x - 2, x**2 + y])
[{x: 2, y: -4}]
>>> solve([x - 2, x**2 + f(x)], {f(x), x})
[{x: 2, f(x): -4}]
```

- if any equation doesn't depend on the symbol(s) given it will be eliminated from the equation set and an answer may be given implicitly in terms of variables that were not of interest

```
>>> solve([x - y, y - 3], x)
{x: y}
```

Disabling High-order, Explicit Solutions

When solving polynomial expressions, one might not want explicit solutions (which can be quite long). If the expression is univariate, CRootOf instances will be returned instead:

```
>>> solve(x**3 - x + 1)
[-1/((-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)) - (-1/2 -
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3, -(-1/2 +
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3 - 1/((-1/2 +
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)), -(3*sqrt(69)/2 +
27/2)**(1/3)/3 - 1/(3*sqrt(69)/2 + 27/2)**(1/3)]
>>> solve(x**3 - x + 1, cubics=False)
[CRootOf(x**3 - x + 1, 0),
 CRootOf(x**3 - x + 1, 1),
 CRootOf(x**3 - x + 1, 2)]
```

If the expression is multivariate, no solution might be returned:

```
>>> solve(x**3 - x + a, x, cubics=False)
[]
```

Sometimes solutions will be obtained even when a flag is False because the expression could be factored. In the following example, the equation can be factored as the product of a linear and a quadratic factor so explicit solutions (which did not require solving a cubic expression) are obtained:

```
>>> eq = x**3 + 3*x**2 + x - 1
>>> solve(eq, cubics=False)
[-1, -1 + sqrt(2), -sqrt(2) - 1]
```

Solving Equations Involving Radicals

Because of SymPy's use of the principle root (issue #8789), some solutions to radical equations will be missed unless `check=False`:

```
>>> from sympy import root
>>> eq = root(x**3 - 3*x**2, 3) + 1 - x
>>> solve(eq)
[]
>>> solve(eq, check=False)
[1/3]
```

In the above example there is only a single solution to the equation. Other expressions will yield spurious roots which must be checked manually; roots which give a negative argument to odd-powered radicals will also need special checking:

```
>>> from sympy import real_root, S
>>> eq = root(x, 3) - root(x, 5) + S(1)/7
>>> solve(eq) # this gives 2 solutions but misses a 3rd
[CRootOf(7*_p**5 - 7*_p**3 + 1, 1)**15,
 CRootOf(7*_p**5 - 7*_p**3 + 1, 2)**15]
>>> sol = solve(eq, check=False)
>>> [abs(eq.subs(x,i).n(2)) for i in sol]
[0.48, 0.e-110, 0.e-110, 0.052, 0.052]
```

The first solution is negative so `real_root` must be used to see that it satisfies the expression:

```
>>> abs(real_root(eq.subs(x, sol[0])).n(2))
0.e-110
```

If the roots of the equation are not real then more care will be necessary to find the roots, especially for higher order equations. Consider the following expression:

```
>>> expr = root(x, 3) - root(x, 5)
```

We will construct a known value for this expression at $x = 3$ by selecting the 1-th root for each radical:

```
>>> expr1 = root(x, 3, 1) - root(x, 5, 1)
>>> v = expr1.subs(x, -3)
```

The `solve` function is unable to find any exact roots to this equation:

```
>>> eq = Eq(expr, v); eq1 = Eq(expr1, v)
>>> solve(eq, check=False), solve(eq1, check=False)
([], [])
```

The function `unrad`, however, can be used to get a form of the equation for which numerical roots can be found:

```
>>> from sympy.solvers.solvers import unrad
>>> from sympy import nroots
>>> e, (p, cov) = unrad(eq)
>>> pvals = nroots(e)
>>> inversion = solve(cov, x)[0]
>>> xvals = [inversion.subs(p, i) for i in pvals]
```

Although `eq` or `eq1` could have been used to find `xvals`, the solution can only be verified with `expr1`:

```
>>> z = expr - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z.subs(x, xi).n()) < 1e-9]
[]
>>> z1 = expr1 - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z1.subs(x, xi).n()) < 1e-9]
[-3.0]
```

`sympy.solvers.solvers.solve_linear(lhs, rhs=0, symbols=[], exclude=[])`

Return a tuple derived from $f = \text{lhs} - \text{rhs}$ that is one of the following:

(0, 1) meaning that f is independent of the symbols in `symbols` that aren't in `exclude`, e.g:

```
>>> from sympy.solvers.solvers import solve_linear
>>> from sympy.abc import x, y, z
>>> from sympy import cos, sin
>>> eq = y*cos(x)**2 + y*sin(x)**2 - y # = y*(1 - 1) = 0
>>> solve_linear(eq)
(0, 1)
>>> eq = cos(x)**2 + sin(x)**2 # = 1
>>> solve_linear(eq)
(0, 1)
```

(continues on next page)

(continued from previous page)

```
>>> solve_linear(x, exclude=[x])  
(0, 1)
```

(0, 0) meaning that there is no solution to the equation amongst the symbols given.

(If the first element of the tuple is not zero then the function is guaranteed to be dependent on a symbol in symbols.)

(symbol, solution) where symbol appears linearly in the numerator of f , is in symbols (if given) and is not in exclude (if given). No simplification is done to f other than a `mul=True` expansion, so the solution will correspond strictly to a unique solution.

(n , d) where n and d are the numerator and denominator of f when the numerator was not linear in any symbol of interest; n will never be a symbol unless a solution for that symbol was found (in which case the second element is the solution, not the denominator).

Examples

```
>>> from sympy.core.power import Pow  
>>> from sympy.polys.polytools import cancel
```

The variable x appears as a linear variable in each of the following:

```
>>> solve_linear(x + y**2)  
(x, -y**2)  
>>> solve_linear(1/x - y**2)  
(x, y**(-2))
```

When not linear in x or y then the numerator and denominator are returned.

```
>>> solve_linear(x**2/y**2 - 3)  
(x**2 - 3*y**2, y**2)
```

If the numerator of the expression is a symbol then (0, 0) is returned if the solution for that symbol would have set any denominator to 0:

```
>>> eq = 1/(1/x - 2)  
>>> eq.as_numer_denom()  
(x, -2*x + 1)  
>>> solve_linear(eq)  
(0, 0)
```

But automatic rewriting may cause a symbol in the denominator to appear in the numerator so a solution will be returned:

```
>>> (1/x)**-1  
x  
>>> solve_linear((1/x)**-1)  
(x, 0)
```

Use an unevaluated expression to avoid this:

```
>>> solve_linear(Pow(1/x, -1, evaluate=False))
(0, 0)
```

If x is allowed to cancel in the following expression, then it appears to be linear in x , but this sort of cancellation is not done by `solve_linear` so the solution will always satisfy the original expression without causing a division by zero error.

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve_linear(cancel(eq))
(x, 0)
>>> solve_linear(eq)
(x**2*(-z**2 + 1), x)
```

A list of symbols for which a solution is desired may be given:

```
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

A list of symbols to ignore may also be given:

```
>>> solve_linear(x + y + z, exclude=[x])
(y, -x - z)
```

(A solution for y is obtained because it is the first variable from the canonically sorted list of symbols that had a linear solution.)

`sympy.solvers.solvers.solve_linear_system(system, *symbols, **flags)`

Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported. The possible number of solutions is zero, one or infinite. Respectively, this procedure will return `None` or a dictionary with solutions. In the case of underdetermined systems, all arbitrary parameters are skipped. This may cause a situation in which an empty dictionary is returned. In that case, all symbols can be assigned arbitrary values.

Input to this functions is a $N \times (M+1)$ matrix, which means it has to be in augmented form. If you prefer to enter N equations and M unknowns then use `solve(Neqs, *Msymbols)` instead. Note: a local copy of the matrix is made by this routine so the matrix that is passed will not be modified.

The algorithm used here is fraction-free Gaussian elimination, which results, after elimination, in an upper-triangular matrix. Then solutions are found using back-substitution. This approach is more efficient and compact than the Gauss-Jordan method.

```
>>> from sympy import Matrix, solve_linear_system
>>> from sympy.abc import x, y
```

Solve the following system:

```
  x + 4 y == 2
-2 x +   y == 14
```

```
>>> system = Matrix(( (1, 4, 2), (-2, 1, 14)))
>>> solve_linear_system(system, x, y)
{x: -6, y: 2}
```

A degenerate system returns an empty dictionary.

```
>>> system = Matrix(( (0,0,0), (0,0,0) ))
>>> solve_linear_system(system, x, y)
{}
```

`sympy.solvers.solvers.solve_linear_system_LU(matrix, syms)`

Solves the augmented matrix system using LUsolve and returns a dictionary in which solutions are keyed to the symbols of syms as ordered.

The matrix must be invertible.

See also:

`sympy.matrices.LUsolve`

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.solvers import solve_linear_system_LU
```

```
>>> solve_linear_system_LU(Matrix([
... [1, 2, 0, 1],
... [3, 2, 2, 1],
... [2, 0, 0, 1]]), [x, y, z])
{x: 1/2, y: 1/4, z: -1/2}
```

`sympy.solvers.solvers.solve_undetermined_coeffs(equ, coeffs, sym, **flags)`

Solve equation of a type $p(x; a_1, \dots, a_k) = q(x)$ where both p, q are univariate polynomials and f depends on k parameters. The result of this functions is a dictionary with symbolic values of those parameters with respect to coefficients in q .

This functions accepts both Equations class instances and ordinary SymPy expressions. Specification of parameters and variable is obligatory for efficiency and simplicity reason.

```
>>> from sympy import Eq
>>> from sympy.abc import a, b, c, x
>>> from sympy.solvers import solve_undetermined_coeffs
```

```
>>> solve_undetermined_coeffs(Eq(2*a*x + a+b, x), [a, b], x)
{a: 1/2, b: -1/2}
```

```
>>> solve_undetermined_coeffs(Eq(a*c*x + a+b, x), [a, b], x)
{a: 1/c, b: -1/c}
```

`sympy.solvers.solvers.nsolve(*args, **kwargs)`

Solve a nonlinear equation system numerically:

```
nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)
```

f is a vector function of symbolic expressions representing the system. $args$ are the variables. If there is only one variable, this argument can be omitted. $x0$ is a starting vector close to a solution.

Use the `modules` keyword to specify which modules should be used to evaluate the function and the Jacobian matrix. Make sure to use a module that supports matrices. For more information on the syntax, please see the docstring of `lambdify`.

If the keyword arguments contain `'dict'=True` (default is `False`) `nsolve` will return a list (perhaps empty) of solution mappings. This might be especially useful if you want to use `nsolve` as a fallback to `solve` since using the `dict` argument for both methods produces return values of consistent type structure. Please note: to keep this consistency with `solve`, the solution will be returned in a list even though `nsolve` (currently at least) only finds one solution at a time.

Overdetermined systems are supported.

```
>>> from sympy import Symbol, nsolve
>>> import sympy
>>> import mpmath
>>> mpmath.mp.dps = 15
>>> x1 = Symbol('x1')
>>> x2 = Symbol('x2')
>>> f1 = 3 * x1**2 - 2 * x2**2 - 1
>>> f2 = x1**2 - 2 * x1 + x2**2 + 2 * x2 - 8
>>> print(nsolve((f1, f2), (x1, x2), (-1, 1)))
Matrix([[ -1.19287309935246], [1.27844411169911]])
```

For one-dimensional functions the syntax is simplified:

```
>>> from sympy import sin, nsolve
>>> from sympy.abc import x
>>> nsolve(sin(x), x, 2)
3.14159265358979
>>> nsolve(sin(x), 2)
3.14159265358979
```

To solve with higher precision than the default, use the `prec` argument.

```
>>> from sympy import cos
>>> nsolve(cos(x) - x, 1)
0.739085133215161
>>> nsolve(cos(x) - x, 1, prec=50)
0.73908513321516064165531208767387340401341175890076
>>> cos(_)
0.73908513321516064165531208767387340401341175890076
```

To solve for complex roots of real functions, a nonreal initial point must be specified:

```
>>> from sympy import I
>>> nsolve(x**2 + 2, I)
1.4142135623731*I
```

`mpmath.findroot` is used and you can find there more extensive documentation, especially concerning keyword parameters and available solvers. Note, however, that functions which are very steep near the root the verification of the solution may fail. In this case you should use the flag `verify = False` and independently verify the solution.

```
>>> from sympy import cos, cosh
>>> from sympy.abc import i
>>> f = cos(x)*cosh(x) - 1
>>> nsolve(f, 3.14*100)
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (1.39267e+230 > 2.1684e-
↪ 19)
```

(continues on next page)

(continued from previous page)

```
>>> ans = nsolve(f, 3.14*100, verify=False); ans
312.588469032184
>>> f.subs(x, ans).n(2)
2.1e+121
>>> (f/f.diff(x)).subs(x, ans).n(2)
7.4e-15
```

One might safely skip the verification if bounds of the root are known and a bisection method is used:

```
>>> bounds = lambda i: (3.14*i, 3.14*(i + 1))
>>> nsolve(f, bounds(100), solver='bisect', verify=False)
315.730061685774
```

Alternatively, a function may be better behaved when the denominator is ignored. Since this is not always the case, however, the decision of what function to use is left to the discretion of the user.

```
>>> eq = x**2/(1 - x)/(1 - 2*x)**2 - 100
>>> nsolve(eq, 0.46)
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (10000 > 2.1684e-19)
Try another starting point or tweak arguments.
>>> nsolve(eq.as_numer_denom()[0], 0.46)
0.46792545969349058
```

`sympy.solvers.solvers.check_assumptions(expr, against=None, **assumptions)`
Checks whether expression *expr* satisfies all assumptions.

assumptions is a dict of assumptions: {'assumption': True|False, ...}.

See also:

`failing_assumptions`

Examples

```
>>> from sympy import Symbol, pi, I, exp, check_assumptions
```

```
>>> check_assumptions(-5, integer=True)
True
>>> check_assumptions(pi, real=True, integer=False)
True
>>> check_assumptions(pi, real=True, negative=True)
False
>>> check_assumptions(exp(I*pi/7), real=False)
True
```

```
>>> x = Symbol('x', real=True, positive=True)
>>> check_assumptions(2*x + 1, real=True, positive=True)
True
>>> check_assumptions(-2*x - 5, real=True, positive=True)
False
```

To check assumptions of `expr` against another variable or expression, pass the expression or variable as `against`.

```
>>> check_assumptions(2*x + 1, x)
True
```

`None` is returned if `check_assumptions()` could not conclude.

```
>>> check_assumptions(2*x - 1, real=True, positive=True)
>>> z = Symbol('z')
>>> check_assumptions(z, real=True)
```

`sympy.solvers.solvers.checksol(f, symbol, sol=None, **flags)`

Checks whether `sol` is a solution of equation `f == 0`.

Input can be either a single symbol and corresponding value or a dictionary of symbols and values. When given as a dictionary and flag `simplify=True`, the values in the dictionary will be simplified. `f` can be a single equation or an iterable of equations. A solution must satisfy all equations in `f` to be considered valid; if a solution does not satisfy any equation, `False` is returned; if one or more checks are inconclusive (and none are `False`) then `None` is returned.

Examples

```
>>> from sympy import symbols
>>> from sympy.solvers import checksol
>>> x, y = symbols('x,y')
>>> checksol(x**4 - 1, x, 1)
True
>>> checksol(x**4 - 1, x, 0)
False
>>> checksol(x**2 + y**2 - 5**2, {x: 3, y: 4})
True
```

To check if an expression is zero using `checksol`, pass it as `f` and send an empty dictionary for `symbol`:

```
>>> checksol(x**2 + x - x*(x + 1), {})
True
```

`None` is returned if `checksol()` could not conclude.

flags:

'numerical=True (default)' do a fast numerical check if `f` has only one symbol.

'minimal=True (default is False)' a very fast, minimal testing.

'warn=True (default is False)' show a warning if `checksol()` could not conclude.

'simplify=True (default)' simplify solution before substituting into function and simplify the function before trying specific simplifications

'force=True (default is False)' make positive all symbols without assumptions regarding sign.

5.31.2 Ordinary Differential equations (ODEs)

See [ODE](#) (page 1230).

5.31.3 Partial Differential Equations (PDEs)

See [PDE](#) (page 1287).

5.31.4 Deutils (Utilities for solving ODE's and PDE's)

`sympy.solvers.deutils.ode_order(expr, func)`

Returns the order of a given differential equation with respect to `func`.

This function is implemented recursively.

Examples

```
>>> from sympy import Function
>>> from sympy.solvers.deutils import ode_order
>>> from sympy.abc import x
>>> f, g = map(Function, ['f', 'g'])
>>> ode_order(f(x).diff(x, 2) + f(x).diff(x)**2 +
... f(x).diff(x), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), g(x))
3
```

5.31.5 Recurrence Equations

`sympy.solvers.recurr.rsolve(f, y, init=None)`

Solve univariate recurrence with rational coefficients.

Given k -th order linear recurrence $Ly = f$, or equivalently:

$$a_k(n)y(n+k) + a_{k-1}(n)y(n+k-1) + \cdots + a_0(n)y(n) = f(n)$$

where $a_i(n)$, for $i = 0, \dots, k$, are polynomials or rational functions in n , and f is a hypergeometric function or a sum of a fixed number of pairwise dissimilar hypergeometric terms in n , finds all solutions or returns `None`, if none were found.

Initial conditions can be given as a dictionary in two forms:

1. `{ n_0 : v_0, n_1 : v_1, ..., n_m : v_m }`
2. `{ y(n_0) : v_0, y(n_1) : v_1, ..., y(n_m) : v_m }`

or as a list `L` of values:

`L = [v_0, v_1, ..., v_m]`

where `L[i] = v_i`, for $i = 0, \dots, m$, maps to $y(n_i)$.

See also:

[rsolve_poly](#) (page 1311), [rsolve_ratio](#) (page 1311), [rsolve_hyper](#) (page 1312)

Examples

Lets consider the following recurrence:

$$(n-1)y(n+2) - (n^2 + 3n - 2)y(n+1) + 2n(n+1)y(n) = 0$$

```
>>> from sympy import Function, rsolve
>>> from sympy.abc import n
>>> y = Function('y')
```

```
>>> f = (n - 1)*y(n + 2) - (n**2 + 3*n - 2)*y(n + 1) + 2*n*(n + 1)*y(n)
```

```
>>> rsolve(f, y(n))
2**n*C0 + C1*factorial(n)
```

```
>>> rsolve(f, y(n), {y(0):0, y(1):3})
3*2**n - 3*factorial(n)
```

`sympy.solvers.recurr.resolve_poly(coeffs, f, n, **hints)`

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$, where f is a polynomial, we seek for all polynomial solutions over field K of characteristic zero.

The algorithm performs two basic steps:

1. Compute degree N of the general polynomial solution.
2. Find all polynomials of degree N or less of $Ly = f$.

There are two methods for computing the polynomial solutions. If the degree bound is relatively small, i.e. it's smaller than or equal to the order of the recurrence, then naive method of undetermined coefficients is being used. This gives system of algebraic equations with $N + 1$ unknowns.

In the other case, the algorithm performs transformation of the initial equation to an equivalent one, for which the system of algebraic equations has only r indeterminates. This method is quite sophisticated (in comparison with the naive one) and was invented together by Abramov, Bronstein and Petkovsek.

It is possible to generalize the algorithm implemented here to the case of linear q -difference and differential equations.

Lets say that we would like to compute m -th Bernoulli polynomial up to a constant. For this we can use $b(n+1) - b(n) = mn^{m-1}$ recurrence, which has solution $b(n) = B_m + C$. For example:

```
>>> from sympy import Symbol, resolve_poly
>>> n = Symbol('n', integer=True)
```

```
>>> resolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

References

[R492] (page 1870), [R493] (page 1870), [R494] (page 1870)

`sympy.solvers.recurr.rsolve_ratio`(coeffs, f, n, **hints)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$, where f is a polynomial, we seek for all rational solutions over field K of characteristic zero.

This procedure accepts only polynomials, however if you are interested in solving recurrence with rational coefficients then use `rsolve` which will pre-process the given equation and run this procedure with polynomial arguments.

The algorithm performs two basic steps:

1. Compute polynomial $v(n)$ which can be used as universal denominator of any rational solution of equation $Ly = f$.
2. Construct new linear difference equation by substitution $y(n) = u(n)/v(n)$ and solve it for $u(n)$ finding all its polynomial solutions. Return `None` if none were found.

Algorithm implemented here is a revised version of the original Abramov's algorithm, developed in 1989. The new approach is much simpler to implement and has better overall efficiency. This method can be easily adapted to q -difference equations case.

Besides finding rational solutions alone, this functions is an important part of Hyper algorithm were it is used to find particular solution of inhomogeneous part of a recurrence.

See also:

[rsolve_hyper](#) (page 1312)

References

[R495] (page 1870)

Examples

```
>>> from sympy.abc import x
>>> from sympy.solvers.recurr import rsolve_ratio
>>> rsolve_ratio([-2*x**3 + x**2 + 2*x - 1, 2*x**3 + x**2 - 6*x,
... - 2*x**3 - 11*x**2 - 18*x - 9, 2*x**3 + 13*x**2 + 22*x + 8], 0, x)
C2*(2*x - 3)/(2*(x**2 - 1))
```

`sympy.solvers.recurr.rsolve_hyper`(coeffs, f, n, **hints)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$ we seek for all hypergeometric solutions over field K of characteristic zero.

The inhomogeneous part can be either hypergeometric or a sum of a fixed number of pairwise dissimilar hypergeometric terms.

The algorithm performs three basic steps:

1. Group together similar hypergeometric terms in the inhomogeneous part of $Ly = f$, and find particular solution using Abramov's algorithm.
2. Compute generating set of L and find basis in it, so that all solutions are linearly independent.
3. Form final solution with the number of arbitrary constants equal to dimension of basis of L .

Term $a(n)$ is hypergeometric if it is annihilated by first order linear difference equations with polynomial coefficients or, in simpler words, if consecutive term ratio is a rational function.

The output of this procedure is a linear combination of fixed number of hypergeometric terms. However the underlying method can generate larger class of solutions - D'Alembertian terms.

Note also that this method not only computes the kernel of the inhomogeneous equation, but also reduces in to a basis so that solutions generated by this procedure are linearly independent

References

[R496] (page 1870), [R497] (page 1871)

Examples

```
>>> from sympy.solvers import rsolve_hyper
>>> from sympy.abc import x
```

```
>>> rsolve_hyper([-1, -1, 1], 0, x)
C0*(1/2 + sqrt(5)/2)**x + C1*(-sqrt(5)/2 + 1/2)**x
```

```
>>> rsolve_hyper([-1, 1], 1 + x, x)
C0 + x*(x + 1)/2
```

5.31.6 Systems of Polynomial Equations

`sympy.solvers.polysys.solve_poly_system(seq, *gens, **args)`
Solve a system of polynomial equations.

Examples

```
>>> from sympy import solve_poly_system
>>> from sympy.abc import x, y
```

```
>>> solve_poly_system([x*y - 2*y, 2*y**2 - x**2], x, y)
[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]
```

`sympy.solvers.polysys.solve_triangulated(polys, *gens, **args)`
Solve a polynomial system using Gianni-Kalkbrenner algorithm.

The algorithm proceeds by computing one Groebner basis in the ground domain and then by iteratively computing polynomial factorizations in appropriately constructed algebraic extensions of the ground domain.

References

1. Patrizia Gianni, Teo Mora, Algebraic Solution of System of Polynomial Equations using Groebner Bases, AAECC-5 on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, LNCS 356 247-257, 1989

Examples

```
>>> from sympy.solvers.polysys import solve_triangulated
>>> from sympy.abc import x, y, z
```

```
>>> F = [x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1]
```

```
>>> solve_triangulated(F, x, y, z)
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
```

5.31.7 Diophantine Equations (DEs)

See [Diophantine](#) (page 1314)

5.31.8 Inequalities

See [Inequality Solvers](#) (page 1339)

5.32 Diophantine

5.32.1 Diophantine equations

The word “Diophantine” comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the “father of Algebra”, Diophantus in his famous work “Arithmetica” presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a “Diophantine equation” to an equation of the form, $f(x_1, x_2, \dots, x_n) = 0$ where $n \geq 2$ and $x_1, x_2, \dots, x_n$ are integer variables. If we can find n integers $a_1, a_2, \dots, a_n$ such that $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$ satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in¹ and².

Currently, following five types of Diophantine equations can be solved using `diophantine()` (page 1319) and other helper functions of the Diophantine module.

- Linear Diophantine equations: $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$.
- General binary quadratic equation: $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation: $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$

¹ Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations

² Diophantine Equation, Wolfram Mathworld, [online]. Available: <http://mathworld.wolfram.com/DiophantineEquation.html>

- Extended Pythagorean equation: $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
- General sum of squares: $x_1^2 + x_2^2 + \dots + x_n^2 = k$

5.32.2 Module structure

This module contains `diophantine()` (page 1319) and helper functions that are needed to solve certain Diophantine equations. It's structured in the following manner.

- `diophantine()` (page 1319)
 - `diop_solve()` (page 1321)
 - * `classify_diop()` (page 1320)
 - * `diop_linear()` (page 1321)
 - * `diop_quadratic()` (page 1323)
 - * `diop_ternary_quadratic()` (page 1327)
 - * `diop_ternary_quadratic_normal()` (page 1336)
 - * `diop_general_pythagorean()` (page 1329)
 - * `diop_general_sum_of_squares()` (page 1329)
 - * `diop_general_sum_of_even_powers()` (page 1330)
 - `merge_solution()` (page 1334)

When an equation is given to `diophantine()` (page 1319), it factors the equation(if possible) and solves the equation given by each factor by calling `diop_solve()` (page 1321) separately. Then all the results are combined using `merge_solution()` (page 1334).

`diop_solve()` (page 1321) internally uses `classify_diop()` (page 1320) to find the type of the equation(and some other details) given to it and then calls the appropriate solver function based on the type returned. For example, if `classify_diop()` (page 1320) returned "linear" as the type of the equation, then `diop_solve()` (page 1321) calls `diop_linear()` (page 1321) to solve the equation.

Each of the functions, `diop_linear()` (page 1321), `diop_quadratic()` (page 1323), `diop_ternary_quadratic()` (page 1327), `diop_general_pythagorean()` (page 1329) and `diop_general_sum_of_squares()` (page 1329) solves a specific type of equations and the type can be easily guessed by it's name.

Apart from these functions, there are a considerable number of other functions in the "Diophantine Module" and all of them are listed under User functions and Internal functions.

5.32.3 Tutorial

First, let's import the highest API of the Diophantine module.

```
>>> from sympy.solvers.diophantine import diophantine
```

Before we start solving the equations, we need to define the variables.

```
>>> from sympy import symbols
>>> x, y, z = symbols("x, y, z", integer=True)
```

Let's start by solving the easiest type of Diophantine equations, i.e. linear Diophantine equations. Let's solve $2x + 3y = 5$. Note that although we write the equation in the above form, when we input the equation to any of the functions in Diophantine module, it needs to be in the form $eq = 0$.

```
>>> diophantine(2*x + 3*y - 5)
{(3*t_0 - 5, -2*t_0 + 5)}
```

Note that stepping one more level below the highest API, we can solve the very same equation by calling `diop_solve()` (page 1321).

```
>>> from sympy.solvers.diophantine import diop_solve
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
```

Note that it returns a tuple rather than a set. `diophantine()` (page 1319) always return a set of tuples. But `diop_solve()` (page 1321) may return a single tuple or a set of tuples depending on the type of the equation given.

We can also solve this equation by calling `diop_linear()` (page 1321), which is what `diop_solve()` (page 1321) calls internally.

```
>>> from sympy.solvers.diophantine import diop_linear
>>> diop_linear(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
```

If the given equation has no solutions then the outputs will look like below.

```
>>> diophantine(2*x + 4*y - 3)
set()
>>> diop_solve(2*x + 4*y - 3)
(None, None)
>>> diop_linear(2*x + 4*y - 3)
(None, None)
```

Note that except for the highest level API, in case of no solutions, a tuple of *None* are returned. Size of the tuple is the same as the number of variables. Also, one can specifically set the parameter to be used in the solutions by passing a customized parameter. Consider the following example:

```
>>> m = symbols("m", integer=True)
>>> diop_solve(2*x + 3*y - 5, m)
(3*m_0 - 5, -2*m_0 + 5)
```

For linear Diophantine equations, the customized parameter is the prefix used for each free variable in the solution. Consider the following example:

```
>>> diop_solve(2*x + 3*y - 5*z + 7, m)
(m_0, m_0 + 5*m_1 - 14, m_0 + 3*m_1 - 7)
```

In the solution above, `m_0` and `m_1` are independent free variables.

Please note that for the moment, users can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let's try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases

associated with the equation would help a lot. Please refer³ and⁴ for detailed analysis of different cases and the nature of the solutions. Let us define $\Delta = b^2 - 4ac$ w.r.t. the binary quadratic $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

When $\Delta < 0$, there are either no solutions or only a finite number of solutions.

```
>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
{(2, 1), (5, 1)}
```

In the above equation $\Delta = (-4)^2 - 4 * 1 * 8 = -16$ and hence only a finite number of solutions exist.

When $\Delta = 0$ we might have either no solutions or parameterized solutions.

```
>>> diophantine(3*x**2 - 6*x*y + 3*y**2 - 3*x + 7*y - 5)
set()
>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
{(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)}
>>> diophantine(x**2 + 2*x*y + y**2 - 3*x - 3*y)
{(t_0, -t_0), (t_0, -t_0 + 3)}
```

The most interesting case is when $\Delta > 0$ and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where $\Delta = 8$.

```
>>> diophantine(x**2 - 4*x*y + 2*y**2 - 3*x + 7*y - 5)
set()
>>> from sympy import sqrt
>>> n = symbols("n", integer=True)
>>> s = diophantine(x**2 - 2*y**2 - 2*x - 4*y, n)
>>> x_1, y_1 = s.pop()
>>> x_2, y_2 = s.pop()
>>> x_n = -(-2*sqrt(2) + 3)**n/2 + sqrt(2)*(-2*sqrt(2) + 3)**n/2 - sqrt(2)*(2*sqrt(2) +
↳ 3)**n/2 - (2*sqrt(2) + 3)**n/2 + 1
>>> x_1 == x_n or x_2 == x_n
True
>>> y_n = -sqrt(2)*(-2*sqrt(2) + 3)**n/4 + (-2*sqrt(2) + 3)**n/2 + sqrt(2)*(2*sqrt(2) +
↳ 3)**n/4 + (2*sqrt(2) + 3)**n/2 - 1
>>> y_1 == y_n or y_2 == y_n
True
```

Here n is an integer. Although x_n and y_n may not look like integers, substituting in specific values for n (and simplifying) shows that they are. For example consider the following example where we set n equal to 9.

```
>>> from sympy import simplify
>>> simplify(x_n.subs({n: 9}))
-9369318
```

Any binary quadratic of the form $ax^2 + bxy + cy^2 + dx + ey + f = 0$ can be transformed to an equivalent form $X^2 - DY^2 = N$.

```
>>> from sympy.solvers.diophantine import find_DN, diop_DN, transformation_to_DN
>>> find_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
(5, 920)
```

³ Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>

⁴ Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <http://www.jpr2718.org/ax2p.pdf>

So, the above equation is equivalent to the equation $X^2 - 5Y^2 = 920$ after a linear transformation. If we want to find the linear transformation, we can use `transformation_to_DN()` (page 1325)

```
>>> A, B = transformation_to_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
```

Here A is a 2 X 2 matrix and B is a 2 X 1 matrix such that the transformation

$$\begin{bmatrix} X \\ Y \end{bmatrix} = A \begin{bmatrix} x \\ y \end{bmatrix} + B$$

gives the equation $X^2 - 5Y^2 = 920$. Values of A and B are as follows.

```
>>> A
Matrix([
[1/10, 3/10],
[ 0, 1/5]])
>>> B
Matrix([
[ 1/5],
[-11/5]])
```

We can solve an equation of the form $X^2 - DY^2 = N$ by passing D and N to `diop_DN()` (page 1323)

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation has no solution.

Now let's turn to homogeneous ternary quadratic equations. These equations are of the form $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$. These type of equations either have infinitely many solutions or no solutions (except the obvious solution (0, 0, 0))

```
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
{(0, 0, 0)}
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
{(-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + 68*q**2)}
```

If you are only interested in a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use `diop_ternary_quadratic()` (page 1327).

```
>>> from sympy.solvers.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
(-4, 5, 1)
```

`diop_ternary_quadratic()` (page 1327) first converts the given equation to an equivalent equation of the form $w^2 = AX^2 + BY^2$ and then it uses `descent()` (page 1328) to solve the latter equation. You can refer to the docs of `transformation_to_normal()` to find more on this. The equation $w^2 = AX^2 + BY^2$ can be solved more easily by using the Aforementioned `descent()` (page 1328).

```
>>> from sympy.solvers.diophantine import descent
>>> descent(3, 1) # solves the equation w**2 = 3*Y**2 + Z**2
(1, 0, 1)
```

Here the solution tuple is in the order (w, Y, Z)

The extended Pythagorean equation, $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$ and the general sum of squares equation, $x_1^2 + x_2^2 + \dots + x_n^2 = k$ can also be solved using the Diophantine module.

```
>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
{(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5,
 60*t3*t5, 210*t4*t5, 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)}
```

function `diop_general_pythagorean()` (page 1329) can also be called directly to solve the same equation. Either you can call `diop_general_pythagorean()` (page 1329) or use the high level API. For the general sum of squares, this is also true, but one advantage of calling `diop_general_sum_of_squares()` (page 1329) is that you can control how many solutions are returned.

```
>>> from sympy.solvers.diophantine import diop_general_sum_of_squares
>>> eq = a**2 + b**2 + c**2 + d**2 - 18
>>> diophantine(eq)
{(0, 0, 3, 3), (0, 1, 1, 4), (1, 2, 2, 3)}
>>> diop_general_sum_of_squares(eq, 2)
{(0, 0, 3, 3), (1, 2, 2, 3)}
```

The `sum_of_squares()` (page 1333) routine will provide an iterator that returns solutions and one may control whether the solutions contain zeros or not (and the solutions not containing zeros are returned first):

```
>>> from sympy.solvers.diophantine import sum_of_squares
>>> sos = sum_of_squares(18, 4, zeros=True)
>>> next(sos)
(1, 2, 2, 3)
>>> next(sos)
(0, 0, 3, 3)
```

Simple Egyptian fractions can be found with the Diophantine module, too. For example, here are the ways that one might represent $1/2$ as a sum of two unit fractions:

```
>>> from sympy import Eq, S
>>> diophantine(Eq(1/x + 1/y, S(1)/2))
{(-2, 1), (1, -2), (3, 6), (4, 4), (6, 3)}
```

To get a more thorough understanding of the Diophantine module, please refer to the following blog.

<http://thilinaatsympy.wordpress.com/>

5.32.4 References

5.32.5 User Functions

This functions is imported into the global namespace with `from sympy import *`:

`diophantine()`

`sympy.solvers.diophantine.diophantine(eq, param=t, syms=None, permute=False)`

Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.

For example, when solving, $x^2 - y^2 = 0$ this is treated as $(x + y)(x - y) = 0$ and $x + y = 0$ and $x - y = 0$ are solved independently and combined. Each term is solved by calling `diop_solve()`.

Output of `diophantine()` is a set of tuples. The elements of the tuple are the solutions for each variable in the equation and are arranged according to the alphabetic ordering of the variables. e.g. For an equation with two variables, a and b , the first element of the tuple is the solution for a and the second for b .

See also:

`diop_solve` (page 1321), `sympy.utilities.iterables.permute_signs` (page 1430), `sympy.utilities.iterables.signed_permutations` (page 1434)

Examples

```
>>> from sympy.abc import x, y, z
>>> diophantine(x**2 - y**2)
{(t_0, -t_0), (t_0, t_0)}
```

```
>>> diophantine(x*(2*x + 3*y - z))
{(0, n1, n2), (t_0, t_1, 2*t_0 + 3*t_1)}
>>> diophantine(x**2 + 3*x*y + 4*x)
{(0, n1), (3*t_0 - 4, -t_0)}
```

Usage

`diophantine(eq, t, syms)`: Solve the diophantine equation `eq`. `t` is the optional parameter to be used by `diop_solve()`. `syms` is an optional list of symbols which determines the order of the elements in the returned tuple.

By default, only the base solution is returned. If `permute` is set to `True` then permutations of the base solution and/or permutations of the signs of the values will be returned when applicable.

```
>>> from sympy.solvers.diophantine import diophantine
>>> from sympy.abc import a, b
>>> eq = a**4 + b**4 - (2**4 + 3**4)
>>> diophantine(eq)
{(2, 3)}
>>> diophantine(eq, permute=True)
{(-3, -2), (-3, 2), (-2, -3), (-2, 3), (2, -3), (2, 3), (3, -2), (3, 2)}
```

Details

`eq` should be an expression which is assumed to be zero. `t` is the parameter to be used in the solution.

And this function is imported with `from sympy.solvers.diophantine import *`:

`classify_diop()`

`sympy.solvers.diophantine.classify_diop(eq, _dict=True)`

5.32.6 Internal Functions

These functions are intended for internal use in the Diophantine module.

`diop_solve()`

`sympy.solvers.diophantine.diop_solve(eq, param=t)`

Solves the diophantine equation `eq`.

Unlike `diophantine()`, factoring of `eq` is not attempted. Uses `classify_diop()` to determine the type of the equation and calls the appropriate solver function.

See also:

`diophantine` (page 1319)

Examples

```
>>> from sympy.solvers.diophantine import diop_solve
>>> from sympy.abc import x, y, z, w
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(t_0, 8*t_0 + 4*t_1 + 5, 7*t_0 + 3*t_1 + 5)
>>> diop_solve(x + 3*y - 4*z + w - 6)
(t_0, t_0 + t_1, 6*t_0 + 5*t_1 + 4*t_2 - 6, 5*t_0 + 4*t_1 + 3*t_2 - 6)
>>> diop_solve(x**2 + y**2 - 5)
{(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)}
```

Usage

`diop_solve(eq, t)`: Solve diophantine equation, `eq` using `t` as a parameter if needed.

Details

`eq` should be an expression which is assumed to be zero. `t` is a parameter to be used in the solution.

`diop_linear()`

`sympy.solvers.diophantine.diop_linear(eq, param=t)`

Solves linear diophantine equations.

A linear diophantine equation is an equation of the form $a_1x_1 + a_2x_2 + \dots + a_nx_n = 0$ where $a_1, a_2, \dots, a_n$ are integer constants and $x_1, x_2, \dots, x_n$ are integer variables.

See also:

`diop_quadratic` (page 1323), `diop_ternary_quadratic` (page 1327),
`diop_general_pythagorean` (page 1329), `diop_general_sum_of_squares` (page 1329)

Examples

```
>>> from sympy.solvers.diophantine import diop_linear
>>> from sympy.abc import x, y, z, t
>>> diop_linear(2*x - 3*y - 5) # solves equation 2*x - 3*y - 5 == 0
(3*t_0 - 5, 2*t_0 - 5)
```

Here $x = -3*t_0 - 5$ and $y = -2*t_0 - 5$

```
>>> diop_linear(2*x - 3*y - 4*z - 3)
(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)
```

Usage

`diop_linear(eq)`: Returns a tuple containing solutions to the diophantine equation `eq`. Values in the tuple is arranged in the same order as the sorted variables.

Details

`eq` is a linear diophantine equation which is assumed to be zero. `param` is the parameter to be used in the solution.

`base_solution_linear()`

`sympy.solvers.diophantine.base_solution_linear(c, a, b, t=None)`

Return the base solution for the linear equation, $ax + by = c$.

Used by `diop_linear()` to find the base solution of a linear Diophantine equation. If `t` is given then the parametrized solution is returned.

Examples

```
>>> from sympy.solvers.diophantine import base_solution_linear
>>> from sympy.abc import t
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, -2*t + 5)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)
```

Usage

`base_solution_linear(c, a, b, t)`: `a`, `b`, `c` are coefficients in $ax + by = c$ and `t` is the parameter to be used in the solution.

diop_quadratic()

`sympy.solvers.diophantine.diop_quadratic(eq, param=t)`

Solves quadratic diophantine equations.

i.e. equations of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$. Returns a set containing the tuples (x, y) which contains the solutions. If there are no solutions then $(None, None)$ is returned.

See also:

[diop_linear](#) (page 1321), [diop_ternary_quadratic](#) (page 1327), [diop_general_sum_of_squares](#) (page 1329), [diop_general_pythagorean](#) (page 1329)

References

[R469] (page 1871), [R470] (page 1871)

Examples

```
>>> from sympy.abc import x, y, t
>>> from sympy.solvers.diophantine import diop_quadratic
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
{(-1, -1)}
```

Usage

`diop_quadratic(eq, param)`: `eq` is a quadratic binary diophantine equation. `param` is used to indicate the parameter to be used in the solution.

Details

`eq` should be an expression which is assumed to be zero. `param` is a parameter to be used in the solution.

diop_DN()

`sympy.solvers.diophantine.diop_DN(D, N, t=t)`

Solves the equation $x^2 - Dy^2 = N$.

Mainly concerned with the case $D > 0$, D is not a perfect square, which is the same as the generalized Pell equation. The LMM algorithm [R471] (page 1871) is used to solve this equation.

Returns one solution tuple, (x, y) for each class of the solutions. Other solutions of the class can be constructed according to the values of D and N .

See also:

[find_DN](#) (page 1327), [diop_bf_DN](#) (page 1325)

References

[R471] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import diop_DN
>>> diop_DN(13, -4) # Solves equation x**2 - 13*y**2 = -4
[(3, 1), (393, 109), (36, 10)]
```

The output can be interpreted as follows: There are three fundamental solutions to the equation $x^2 - 13y^2 = -4$ given by (3, 1), (393, 109) and (36, 10). Each tuple is in the form (x, y), i.e. solution (3, 1) means that $x = 3$ and $y = 1$.

```
>>> diop_DN(986, 1) # Solves equation x**2 - 986*y**2 = 1
[(49299, 1570)]
```

Usage

`diop_DN(D, N, t)`: D and N are integers as in $x^2 - Dy^2 = N$ and t is the parameter to be used in the solutions.

Details

D and N correspond to D and N in the equation. t is the parameter to be used in the solutions.

cornacchia()

`sympy.solvers.diophantine.cornacchia(a, b, m)`

Solves $ax^2 + by^2 = m$ where $\gcd(a, b) = 1 = \gcd(a, m)$ and $a, b > 0$.

Uses the algorithm due to Cornacchia. The method only finds primitive solutions, i.e. ones with $\gcd(x, y) = 1$. So this method can't be used to find the solutions of $x^2 + y^2 = 20$ since the only solution to former is $(x, y) = (4, 2)$ and it is not primitive. When $a = b$, only the solutions with $x \leq y$ are found. For more details, see the References.

See also:

`sympy.utilities.iterables.signed_permutations` (page 1434)

References

[R472] (page 1871), [R473] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import cornacchia
>>> cornacchia(2, 3, 35) # equation 2x**2 + 3y**2 = 35
{(2, 3), (4, 1)}
>>> cornacchia(1, 1, 25) # equation x**2 + y**2 = 25
{(4, 3)}
```

diop_bf_DN()

`sympy.solvers.diophantine.diop_bf_DN(D, N, t=t)`

Uses brute force to solve the equation, $x^2 - Dy^2 = N$.

Mainly concerned with the generalized Pell equation which is the case when $D > 0$, D is not a perfect square. For more information on the case refer [R474] (page 1871). Let (t, u) be the minimal positive solution of the equation $x^2 - Dy^2 = 1$. Then this method requires $\sqrt{\frac{|N|}{(t \pm 1)2D}}$ to be small.

See also:

[diop_DN](#) (page 1323)

References

[R474] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import diop_bf_DN
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

Usage

`diop_bf_DN(D, N, t)`: D and N are coefficients in $x^2 - Dy^2 = N$ and t is the parameter to be used in the solutions.

Details

D and N correspond to D and N in the equation. t is the parameter to be used in the solutions.

transformation_to_DN()

`sympy.solvers.diophantine.transformation_to_DN(eq)`

This function transforms general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$ to more easy to deal with $X^2 - DY^2 = N$ form.

This is used to solve the general quadratic equation by transforming it to the latter form. Refer [R475] (page 1871) for more detailed information on the transformation. This function returns a tuple (A, B) where A is a 2 X 2 matrix and B is a 2 X 1 matrix such that,

$$\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$$

See also:

[find_DN](#) (page 1327)

References

[R475] (page 1871)

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine import transformation_to_DN
>>> from sympy.solvers.diophantine import classify_diop
>>> A, B = transformation_to_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
>>> A
Matrix([
[1/26, 3/26],
[  0, 1/13]])
>>> B
Matrix([
[-6/13],
[-4/13]])
```

A, B returned are such that $\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$. Substituting these values for x and y and a bit of simplifying work will give an equation of the form $x^2 - Dy^2 = N$.

```
>>> from sympy.abc import X, Y
>>> from sympy import Matrix, simplify
>>> u = (A*Matrix([X, Y]) + B)[0] # Transformation for x
>>> u
X/26 + 3*Y/26 - 6/13
>>> v = (A*Matrix([X, Y]) + B)[1] # Transformation for y
>>> v
Y/13 - 4/13
```

Next we will substitute these formulas for x and y and do `simplify()`.

```
>>> eq = simplify((x**2 - 3*x*y - y**2 - 2*y + 1).subs(zip((x, y), (u, v))))
>>> eq
X**2/676 - Y**2/52 + 17/13
```

By multiplying the denominator appropriately, we can get a Pell equation in the standard form.

```
>>> eq * 676
X**2 - 13*Y**2 + 884
```

If only the final equation is needed, `find_DN()` can be used.

Usage

`transformation_to_DN(eq)`: where `eq` is the quadratic to be transformed.

`find_DN()`

`sympy.solvers.diophantine.find_DN(eq)`

This function returns a tuple, (D, N) of the simplified form, $x^2 - Dy^2 = N$, corresponding to the general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

Solving the general quadratic is then equivalent to solving the equation $X^2 - DY^2 = N$ and transforming the solutions by using the transformation matrices returned by `transformation_to_DN()`.

See also:

[transformation_to_DN](#) (page 1325)

References

[R476] (page 1871)

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine import find_DN
>>> find_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
(13, -884)
```

Interpretation of the output is that we get $X^2 - 13Y^2 = -884$ after transforming $x^2 - 3xy - y^2 - 2y + 1$ using the transformation returned by `transformation_to_DN()`.

Usage

`find_DN(eq)`: where `eq` is the quadratic to be transformed.

`diop_ternary_quadratic()`

`sympy.solvers.diophantine.diop_ternary_quadratic(eq)`

Solves the general quadratic ternary form, $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

Returns a tuple (x, y, z) which is a base solution for the above equation. If there are no solutions, $(None, None, None)$ is returned.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic(45*x**2 - 7*y**2 - 8*x*y - z**2)
(28, 45, 105)
>>> diop_ternary_quadratic(x**2 - 49*y**2 - z**2 + 13*z*y - 8*x*y)
(9, 1, 5)
```

Usage

`diop_ternary_quadratic(eq)`: Return a tuple containing a basic solution to eq.

Details

eq should be an homogeneous expression of degree two in three variables and it is assumed to be zero.

square_factor()

`sympy.solvers.diophantine.square_factor(a)`

Returns an integer c s.t. $a = c^2k$, $c, k \in \mathbb{Z}$. Here k is square free. a can be given as an integer or a dictionary of factors.

See also:

[sympy.ntheory.factor_.core](#) (page 319)

Examples

```
>>> from sympy.solvers.diophantine import square_factor
>>> square_factor(24)
2
>>> square_factor(-36*3)
6
>>> square_factor(1)
1
>>> square_factor({3: 2, 2: 1, -1: 1}) # -18
3
```

descent()

`sympy.solvers.diophantine.descent(A, B)`

Returns a non-trivial solution, (x, y, z) , to $x^2 = Ay^2 + Bz^2$ using Lagrange's descent method with lattice-reduction. A and B are assumed to be valid for such a solution to exist.

This is faster than the normal Lagrange's descent algorithm because the Gaussian reduction is used.

References

[R477] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import descent
>>> descent(3, 1) # x**2 = 3*y**2 + z**2
(1, 0, 1)
```

$(x, y, z) = (1, 0, 1)$ is a solution to the above equation.

```
>>> descent(41, -113)
(-16, -3, 1)
```

diop_general_pythagorean()

`sympy.solvers.diophantine.diop_general_pythagorean(eq, param=m)`

Solves the general pythagorean equation, $a_1^2 x_1^2 + a_2^2 x_2^2 + \dots + a_n^2 x_n^2 - a_{n+1}^2 x_{n+1}^2 = 0$.

Returns a tuple which contains a parametrized solution to the equation, sorted in the same order as the input variables.

Examples

```
>>> from sympy.solvers.diophantine import diop_general_pythagorean
>>> from sympy.abc import a, b, c, d, e
>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2 + 10*m2**2 + 10*m3**2 - 10*m4**2, 15*m1**2 + 15*m2**2 + 15*m3**2 +
↳ 15*m4**2, 15*m1*m4, 12*m2*m4, 60*m3*m4)
```

Usage

`diop_general_pythagorean(eq, param)`: where `eq` is a general pythagorean equation which is assumed to be zero and `param` is the base parameter used to construct other parameters by subscripting.

diop_general_sum_of_squares()

`sympy.solvers.diophantine.diop_general_sum_of_squares(eq, limit=1)`

Solves the equation $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$.

Returns at most `limit` number of solutions.

Examples

```
>>> from sympy.solvers.diophantine import diop_general_sum_of_squares
>>> from sympy.abc import a, b, c, d, e, f
>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345)
{(15, 22, 22, 24, 24)}
```

Usage

`general_sum_of_squares(eq, limit)` : Here `eq` is an expression which is assumed to be zero. Also, `eq` should be in the form, $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$.

Details

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$ then there will be no solutions. Refer [R478] (page 1871) for more details.

Reference

`diop_general_sum_of_even_powers()`

`sympy.solvers.diophantine.diop_general_sum_of_even_powers(eq, limit=1)`
Solves the equation $x_1^e + x_2^e + \dots + x_n^e - k = 0$ where e is an even, integer power.
Returns at most `limit` number of solutions.

See also:

`power_representation`

Examples

```
>>> from sympy.solvers.diophantine import diop_general_sum_of_even_powers
>>> from sympy.abc import a, b
>>> diop_general_sum_of_even_powers(a**4 + b**4 - (2**4 + 3**4))
{(2, 3)}
```

Usage

`general_sum_of_even_powers(eq, limit)` : Here `eq` is an expression which is assumed to be zero. Also, `eq` should be in the form, $x_1^e + x_2^e + \dots + x_n^e - k = 0$.

`partition()`

`sympy.solvers.diophantine.partition(n, k=None, zeros=False)`
Returns a generator that can be used to generate partitions of an integer n .

A partition of n is a set of positive integers which add up to n . For example, partitions of 3 are 3, 1 + 2, 1 + 1 + 1. A partition is returned as a tuple. If `k` equals `None`, then all

possible partitions are returned irrespective of their size, otherwise only the partitions of size k are returned. If the `zero` parameter is set to `True` then a suitable number of zeros are added at the end of every partition of size less than k .

`zero` parameter is considered only if k is not `None`. When the partitions are over, the last `next()` call throws the `StopIteration` exception, so this function should always be used inside a `try - except` block.

Examples

```
>>> from sympy.solvers.diophantine import partition
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(1, 1, 3)
>>> next(g)
(1, 2, 2)
>>> g = partition(5, 3, zeros=True)
>>> next(g)
(0, 0, 5)
```

Details

`partition(n, k)`: Here n is a positive integer and k is the size of the partition which is also positive integer.

`sum_of_three_squares()`

`sympy.solvers.diophantine.sum_of_three_squares(n)`

Returns a 3-tuple (a, b, c) such that $a^2 + b^2 + c^2 = n$ and $a, b, c \geq 0$.

Returns `None` if $n = 4^a(8m+7)$ for some $a, m \in \mathbb{Z}$. See [R479] (page 1871) for more details.

See also:

`sum_of_squares` (page 1333)

References

[R479] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import sum_of_three_squares
>>> sum_of_three_squares(44542)
(18, 37, 207)
```

Usage

`sum_of_three_squares(n)`: Here n is a non-negative integer.

`sum_of_four_squares()`

`sympy.solvers.diophantine.sum_of_four_squares(n)`

Returns a 4-tuple (a, b, c, d) such that $a^2 + b^2 + c^2 + d^2 = n$.

Here $a, b, c, d \geq 0$.

See also:

[sum_of_squares](#) (page 1333)

References

[R480] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import sum_of_four_squares
>>> sum_of_four_squares(3456)
(8, 8, 32, 48)
>>> sum_of_four_squares(1294585930293)
(0, 1234, 2161, 1137796)
```

Usage

`sum_of_four_squares(n)`: Here n is a non-negative integer.

`sum_of_powers()`

`sympy.solvers.diophantine.sum_of_powers(n, p, k, zeros=False)`

Returns a generator for finding k -tuples of integers, $(n_1, n_2, \dots, n_k)$, such that $n = n_1^p + n_2^p + \dots + n_k^p$.

Examples

```
>>> from sympy.solvers.diophantine import power_representation
```

Represent 1729 as a sum of two cubes:

```
>>> f = power_representation(1729, 3, 2)
>>> next(f)
(9, 10)
>>> next(f)
(1, 12)
```

If the flag `zeros` is `True`, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even p the `permute_sign` function can be used to get all signed values:

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-12, -1)]
```

Usage

`power_representation(n, p, k, zeros)`: Represent non-negative number n as a sum of k p 'th powers. If `zeros` is true, then the solutions is allowed to contain zeros.

sum_of_squares()

`sympy.solvers.diophantine.sum_of_squares(n, k, zeros=False)`

Return a generator that yields the k -tuples of nonnegative values, the squares of which sum to n . If `zeros` is `False` (default) then the solution will not contain zeros. The nonnegative elements of a tuple are sorted.

- If $k == 1$ and n is square, $(n,)$ is returned.
- If $k == 2$ then n can only be written as a sum of squares if every prime in the factorization of n that has the form $4*k + 3$ has an even multiplicity. If n is prime then it can only be written as a sum of two squares if it is in the form $4*k + 1$.
- if $k == 3$ then n can be written as a sum of squares if it does not have the form $4*m*(8*k + 7)$.
- all integers can be written as the sum of 4 squares.
- if $k > 4$ then n can be partitioned and each partition can be written as a sum of 4 squares; if n is not evenly divisible by 4 then n can be written as a sum of squares only if the an additional partition can be written as sum of squares. For example, if $k = 6$ then n is partitioned into two parts, the first being written as a sum of 4 squares and the second being written as a sum of 2 squares - which can only be done if the condition above for $k = 2$ can be met, so this will automatically reject certain partitions of n .

See also:

[sympy.utilities.iterables.signed_permutations](#) (page 1434)

Examples

```
>>> from sympy.solvers.diophantine import sum_of_squares
>>> list(sum_of_squares(25, 2))
[(3, 4)]
>>> list(sum_of_squares(25, 2, True))
[(3, 4), (0, 5)]
>>> list(sum_of_squares(25, 4))
[(1, 2, 2, 4)]
```

merge_solution

`sympy.solvers.diophantine.merge_solution(var, var_t, solution)`

This is used to construct the full solution from the solutions of sub equations.

For example when solving the equation $(x - y)(x^2 + y^2 - z^2) = 0$, solutions for each of the equations $x - y = 0$ and $x^2 + y^2 - z^2$ are found independently. Solutions for $x - y = 0$ are $(x, y) = (t, t)$. But we should introduce a value for z when we output the solution for the original equation. This function converts (t, t) into (t, t, n_1) where n_1 is an integer parameter.

divisible

`sympy.solvers.diophantine.divisible(a, b)`

Returns *True* if a is divisible by b and *False* otherwise.

PQa

`sympy.solvers.diophantine.PQa(P_0, Q_0, D)`

Returns useful information needed to solve the Pell equation.

There are six sequences of integers defined related to the continued fraction representation of $\frac{P}{Q} + \sqrt{D}\frac{Q}{Q}$, namely $\{P_i\}$, $\{Q_i\}$, $\{a_i\}$, $\{A_i\}$, $\{B_i\}$, $\{G_i\}$. `PQa()` Returns these values as a 6-tuple in the same order as mentioned above. Refer [R481] (page 1871) for more detailed information.

References

[R481] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import PQa
>>> pqa = PQa(13, 4, 5) # (13 + sqrt(5))/4
>>> next(pqa) # (P_0, Q_0, a_0, A_0, B_0, G_0)
(13, 4, 3, 3, 1, -1)
>>> next(pqa) # (P_1, Q_1, a_1, A_1, B_1, G_1)
(-1, 1, 1, 4, 1, 3)
```

Usage

`PQa(P_0, Q_0, D)`: P_0 , Q_0 and D are integers corresponding to P_0 , Q_0 and D in the continued fraction $\frac{P_0}{Q_0} + \sqrt{D}$. Also it's assumed that $P_0^2 \equiv D \pmod{|Q_0|}$ and D is square free.

equivalent

`sympy.solvers.diophantine.equivalent(u, v, r, s, D, N)`

Returns True if two solutions (u, v) and (r, s) of $x^2 - Dy^2 = N$ belongs to the same equivalence class and False otherwise.

Two solutions (u, v) and (r, s) to the above equation fall to the same equivalence class iff both $(ur - Dvs)$ and $(us - vr)$ are divisible by N . See reference [R482] (page 1871). No test is performed to test whether (u, v) and (r, s) are actually solutions to the equation. User should take care of this.

References

[R482] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import equivalent
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

Usage

`equivalent(u, v, r, s, D, N)`: (u, v) and (r, s) are two solutions of the equation $x^2 - Dy^2 = N$ and all parameters involved are integers.

parametrize_ternary_quadratic

`sympy.solvers.diophantine.parametrize_ternary_quadratic(eq)`

Returns the parametrized general solution for the ternary quadratic equation eq which has the form $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

References

[R483] (page 1871)

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine import parametrize_ternary_quadratic
>>> parametrize_ternary_quadratic(x**2 + y**2 - z**2)
(2*p*q, p**2 - q**2, p**2 + q**2)
```

Here p and q are two co-prime integers.

```
>>> parametrize_ternary_quadratic(3*x**2 + 2*y**2 - z**2 - 2*x*y + 5*y*z - 7*y*z)
(2*p**2 - 2*p*q - q**2, 2*p**2 + 2*p*q - q**2, 2*p**2 - 2*p*q + 3*q**2)
>>> parametrize_ternary_quadratic(124*x**2 - 30*y**2 - 7729*z**2)
(-1410*p**2 - 363263*q**2, 2700*p**2 + 30916*p*q - 695610*q**2, -60*p**2 +
↳ 5400*p*q + 15458*q**2)
```

diop_ternary_quadratic_normal

`sympy.solvers.diophantine.diop_ternary_quadratic_normal(eq)`

Solves the quadratic ternary diophantine equation, $ax^2 + by^2 + cz^2 = 0$.

Here the coefficients a , b , and c should be non zero. Otherwise the equation will be a quadratic binary or univariate equation. If solvable, returns a tuple (x, y, z) that satisfies the given equation. If the equation does not have integer solutions, $(None, None, None)$ is returned.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine import diop_ternary_quadratic_normal
>>> diop_ternary_quadratic_normal(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic_normal(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic_normal(34*x**2 - 3*y**2 - 301*z**2)
(4, 9, 1)
```

Usage

`diop_ternary_quadratic_normal(eq)`: where eq is an equation of the form $ax^2 + by^2 + cz^2 = 0$.

ldescent

`sympy.solvers.diophantine.ldescent(A, B)`

Return a non-trivial solution to $w^2 = Ax^2 + By^2$ using Lagrange's method; return `None` if there is no such solution. .

Here, $A \neq 0$ and $B \neq 0$ and A and B are square free. Output a tuple (w_0, x_0, y_0) which is a solution to the above equation.

References

[R484] (page 1871), [R485] (page 1871)

Examples

```
>>> from sympy.solvers.diophantine import ldescent
>>> ldescent(1, 1) # w^2 = x^2 + y^2
(1, 1, 0)
>>> ldescent(4, -7) # w^2 = 4x^2 - 7y^2
(2, -1, 0)
```

This means that $x = -1, y = 0$ and $w = 2$ is a solution to the equation $w^2 = 4x^2 - 7y^2$

```
>>> ldescent(5, -1) # w^2 = 5x^2 - y^2
(2, 1, -1)
```

gaussian_reduce

`sympy.solvers.diophantine.gaussian_reduce(w, a, b)`

Returns a reduced solution (x, z) to the congruence $X^2 - aZ^2 \equiv 0 \pmod{b}$ so that $x^2 + |a|z^2$ is minimal.

References

[R486] (page 1871), [R487] (page 1871)

Details

Here w is a solution of the congruence $x^2 \equiv a \pmod{b}$

holzer

`sympy.solvers.diophantine.holzer(x, y, z, a, b, c)`

Simplify the solution (x, y, z) of the equation $ax^2 + by^2 = cz^2$ with $a, b, c > 0$ and $z^2 \geq |ab|$ to a new reduced solution (x', y', z') such that $z'^2 \leq |ab|$.

The algorithm is an interpretation of Mordell's reduction as described on page 8 of Cremona and Rusin's paper [R488] (page 1871) and the work of Mordell in reference [R489] (page 1871).

References

[R488] (page 1871), [R489] (page 1871)

prime_as_sum_of_two_squares

`sympy.solvers.diophantine.prime_as_sum_of_two_squares(p)`

Represent a prime p as a unique sum of two squares; this can only be done if the prime is congruent to 1 mod 4.

See also:

[sum_of_squares](#) (page 1333)

Examples

```
>>> from sympy.solvers.diophantine import prime_as_sum_of_two_squares
>>> prime_as_sum_of_two_squares(7) # can't be done
>>> prime_as_sum_of_two_squares(5)
(1, 2)
```

Reference

sqf_normal

`sympy.solvers.diophantine.sqf_normal(a, b, c, steps=False)`

Return a', b', c' , the coefficients of the square-free normal form of $ax^2 + by^2 + cz^2 = 0$, where a', b', c' are pairwise prime. If *steps* is True then also return three tuples: *sq*, *sqf*, and (a', b', c') where *sq* contains the square factors of a , b and c after removing the $\gcd(a, b, c)$; *sqf* contains the values of a , b and c after removing both the $\gcd(a, b, c)$ and the square factors.

The solutions for $ax^2 + by^2 + cz^2 = 0$ can be recovered from the solutions of $a'x^2 + b'y^2 + c'z^2 = 0$.

See also:

[reconstruct](#) (page 1339)

References

[R491] (page 1872)

Examples

```
>>> from sympy.solvers.diophantine import sqf_normal
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11)
(11, 1, 5)
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11, True)
((3, 1, 7), (5, 55, 11), (11, 1, 5))
```


reconstruct

`sympy.solvers.diophantine.reconstruct(A, B, z)`

Reconstruct the z value of an equivalent solution of $ax^2 + by^2 + cz^2$ from the z value of a solution of the square-free normal form of the equation, $a' * x^2 + b' * y^2 + c' * z^2$, where a' , b' and c' are square free and $\gcd(a', b', c') == 1$.

5.33 Inequality Solvers

`sympy.solvers.inequalities.solve_rational_inequalities(eqs)`

Solve a system of rational inequalities with rational coefficients.

See also:

`solve_poly_inequality` (page 1339)

Examples

```
>>> from sympy.abc import x
>>> from sympy import Poly
>>> from sympy.solvers.inequalities import solve_rational_inequalities
```

```
>>> solve_rational_inequalities([[
... ((Poly(-x + 1), Poly(1, x)), '>='),
... ((Poly(-x + 1), Poly(1, x)), '<=')]])
{1}
```

```
>>> solve_rational_inequalities([[
... ((Poly(x), Poly(1, x)), '!='),
... ((Poly(-x + 1), Poly(1, x)), '>=' )]])
Union(Interval.open(-oo, 0), Interval.Lopen(0, 1))
```

`sympy.solvers.inequalities.solve_poly_inequality(poly, rel)`

Solve a polynomial inequality with rational coefficients.

See also:

`solve_poly_inequalities`

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
>>> from sympy.solvers.inequalities import solve_poly_inequality
```

```
>>> solve_poly_inequality(Poly(x, x, domain='ZZ'), '==')
[{0}]
```

```
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '!=')
[Interval.open(-oo, -1), Interval.open(-1, 1), Interval.open(1, oo)]
```

```
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '==')
[{-1}, {1}]
```

`sympy.solvers.inequalities.reduce_rational_inequalities`(*exprs*, *gen*, *relational=True*)
Reduce a system of rational inequalities with rational coefficients.

Examples

```
>>> from sympy import Poly, Symbol
>>> from sympy.solvers.inequalities import reduce_rational_inequalities
```

```
>>> x = Symbol('x', real=True)
```

```
>>> reduce_rational_inequalities([[x**2 <= 0]], x)
Eq(x, 0)
```

```
>>> reduce_rational_inequalities([[x + 2 > 0]], x)
(-2 < x) & (x < oo)
>>> reduce_rational_inequalities([(x + 2, ">")], x)
(-2 < x) & (x < oo)
>>> reduce_rational_inequalities([[x + 2]], x)
Eq(x, -2)
```

`sympy.solvers.inequalities.reduce_abs_inequality`(*expr*, *rel*, *gen*)
Reduce an inequality with nested absolute values.

See also:

[reduce_abs_inequalities](#) (page 1340)

Examples

```
>>> from sympy import Abs, Symbol
>>> from sympy.solvers.inequalities import reduce_abs_inequality
>>> x = Symbol('x', real=True)
```

```
>>> reduce_abs_inequality(Abs(x - 5) - 3, '<', x)
(2 < x) & (x < 8)
```

```
>>> reduce_abs_inequality(Abs(x + 2)*3 - 13, '<', x)
(-19/3 < x) & (x < 7/3)
```

`sympy.solvers.inequalities.reduce_abs_inequalities`(*exprs*, *gen*)
Reduce a system of inequalities with nested absolute values.

See also:

[reduce_abs_inequality](#) (page 1340)

Examples

```
>>> from sympy import Abs, Symbol
>>> from sympy.abc import x
>>> from sympy.solvers.inequalities import reduce_abs_inequalities
>>> x = Symbol('x', real=True)
```

```
>>> reduce_abs_inequalities([(Abs(3*x - 5) - 7, '<'),
... (Abs(x + 25) - 13, '>')], x)
(-2/3 < x) & (x < 4) & (((-oo < x) & (x < -38)) | ((-12 < x) & (x < oo)))
```

```
>>> reduce_abs_inequalities([(Abs(x - 4) + Abs(3*x - 5) - 7, '<')], x)
(1/2 < x) & (x < 4)
```

`sympy.solvers.inequalities.reduce_inequalities(inequalities, symbols=[])`
Reduce a system of inequalities with rational coefficients.

Examples

```
>>> from sympy import sympify as S, Symbol
>>> from sympy.abc import x, y
>>> from sympy.solvers.inequalities import reduce_inequalities
```

```
>>> reduce_inequalities(0 <= x + 3, [])
(-3 <= x) & (x < oo)
```

```
>>> reduce_inequalities(0 <= x + y*2 - 1, [x])
(x < oo) & (x >= -2*y + 1)
```

`sympy.solvers.inequalities.solve_univariate_inequality(expr, gen, relational=True, domain=Reals, continuous=False)`

Solves a real univariate inequality.

Parameters **expr** : Relational

The target inequality

gen : Symbol

The variable for which the inequality is solved

relational : bool

A Relational type output is expected or not

domain : Set

The domain over which the equation is solved

continuous: bool

True if `expr` is known to be continuous over the given domain (and so `continuous_domain()` doesn't need to be called on it)

Raises **NotImplementedError**

The solution of the inequality cannot be determined due to limitation in *sympify*.

See also:

solvify solver returning solveset solutions with solve's output API

Notes

Currently, we cannot solve all the inequalities due to limitations in *solvify*. Also, the solution returned for trigonometric inequalities are restricted in its periodic interval.

Examples

```
>>> from sympy.solvers.inequalities import solve_univariate_inequality
>>> from sympy import Symbol, sin, Interval, S
>>> x = Symbol('x')
```

```
>>> solve_univariate_inequality(x**2 >= 4, x)
((2 <= x) & (x < oo)) | ((x <= -2) & (-oo < x))
```

```
>>> solve_univariate_inequality(x**2 >= 4, x, relational=False)
Union(Interval(-oo, -2), Interval(2, oo))
```

```
>>> domain = Interval(0, S.Infinity)
>>> solve_univariate_inequality(x**2 >= 4, x, False, domain)
Interval(2, oo)
```

```
>>> solve_univariate_inequality(sin(x) > 0, x, relational=False)
Interval.open(0, pi)
```

5.34 Solveset

This is the official documentation of the `solveset` module in solvers. It contains the frequently asked questions about our new module to solve equations.

5.34.1 What's wrong with solve():

SymPy already has a pretty powerful `solve` function. But it has a lot of major issues

1. It doesn't have a consistent output for various types of solutions It needs to return a lot of types of solutions consistently:

- Single solution : $x = 1$
- Multiple solutions: $x^2 = 1$
- No Solution: $x^2 + 1 = 0; x \in \mathbb{R}$
- Interval of solution: $\lfloor x \rfloor = 0$
- Infinitely many solutions: $\sin(x) = 0$
- Multivariate functions with point solutions: $x^2 + y^2 = 0$
- Multivariate functions with non-point solution: $x^2 + y^2 = 1$

- System of equations: $x + y = 1$ and $x - y = 0$
 - Relational: $x > 0$
 - And the most important case: “We don’t Know”
2. The input API is also a mess, there are a lot of parameters. Many of them are not needed and they make it hard for the user and the developers to work on solvers.
 3. There are cases like finding the maxima and minima of function using critical points where it is important to know if it has returned all the solutions. `solve` does not guarantee this.

5.34.2 Why Solveset?

- `solveset` has a cleaner input and output interface: `solveset` returns a set object and a set object takes care of all types of output. For cases where it doesn’t “know” all the solutions a `ConditionSet` with a partial solution is returned. For input it only takes the equation, the variables to solve for and the optional argument `domain` over which the equation is to be solved.
- `solveset` can return infinitely many solutions. For example solving for $\sin(x) = 0$ returns $\{2n\pi | n \in \mathbb{Z}\} \cup \{2n\pi + \pi | n \in \mathbb{Z}\}$, whereas `solve` only returns $[0, \pi]$.
- There is a clear code level and interface level separation between solvers for equations in the complex domain and the real domain. For example solving $e^x = 1$ when x is to be solved in the complex domain, returns the set of all solutions, that is $\{2ni\pi | n \in \mathbb{Z}\}$, whereas if x is to be solved in the real domain then only $\{0\}$ is returned.

5.34.3 Why do we use Sets as an output type?

SymPy has a well developed sets module, which can represent most of the set containers in Mathematics such as:

- `FiniteSet`
Represents a finite set of discrete numbers.
- `Interval`
Represents a real interval as a set.
- `ProductSet`
Represents a Cartesian product of sets.
- `ImageSet`
Represents the image of a set under a mathematical function

```
>>> from sympy import ImageSet, S, Lambda
>>> from sympy.abc import x
>>> squares = ImageSet(Lambda(x, x**2), S.Naturals) # {x**2 for x in N}
>>> 4 in squares
True
```

- `ComplexRegion`
Represents the set of all complex numbers in a region in the Argand plane.
- `ConditionSet`
Represents the set of elements, which satisfies a given condition.

Also, the predefined set classes such as:

- `Naturals` $\mathbb{N}$
Represents the natural numbers (or counting numbers), which are all positive integers starting from 1.
- `Naturals0` $\mathbb{N}_0$
Represents the whole numbers, which are all the non-negative integers, inclusive of 0.
- `Integers` $\mathbb{Z}$
Represents all integers: positive, negative and zero.
- `Reals` $\mathbb{R}$
Represents the set of all real numbers.
- `Complexes` $\mathbb{C}$
Represents the set of all complex numbers.
- `EmptySet` ϕ
Represents the empty set.

The above six sets are available as Singletons, like `S.Integers`.

It is capable of most of the set operations in mathematics:

- `Union`
- `Intersection`
- `Complement`
- `SymmetricDifference`

The main reason for using sets as output to solvers is that it can consistently represent many types of solutions. For the single variable case it can represent:

- No solution (by the empty set).
- Finitely many solutions (by `FiniteSet`).
- Infinitely many solutions, both countably and uncountably infinite solutions (using the `ImageSet` module).
- `Interval`
- There can also be bizarre solutions to equations like the set of rational numbers.

No other Python object (list, dictionary, generator, Python sets) provides the flexibility of mathematical sets which our sets module tries to emulate. The second reason to use sets is that they are close to the entities which mathematicians deal with and it makes it easier to reason about them. Set objects conform to Pythonic conventions when possible, i.e., $x \in A$ and `for i in A` both work when they can be computed. Another advantage of using objects closer to mathematical entities is that the user won't have to "learn" our representation and she can have her expectations transferred from her mathematical experience.

For the multivariate case we represent solutions as a set of points in a n -dimensional space and a point is represented by a `FiniteSet` of ordered tuples, which is a point in $\mathbb{R}^n$ or $\mathbb{C}^n$.

Please note that, the general `FiniteSet` is unordered, but a `FiniteSet` with a tuple as its only argument becomes ordered, since a tuple is ordered. So the order in the tuple is mapped to a pre-defined order of variables while returning solutions.

For example:

```
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3) # Unordered
{1, 2, 3}
>>> FiniteSet((1, 2, 3)) # Ordered
{(1, 2, 3)}
```

Why not use dicts as output?

Dictionary are easy to deal with programmatically but mathematically they are not very precise and use of them can quickly lead to inconsistency and a lot of confusion. For example:

- There are a lot of cases where we don't know the complete solution and we may like to output a partial solution, consider the equation $fg = 0$. The solution of this equation is the union of the solution of the following two equations: $f = 0$, $g = 0$. Let's say that we are able to solve $f = 0$ but solving $g = 0$ isn't supported yet. In this case we cannot represent partial solution of the given equation $fg = 0$ using dicts. This problem is solved with sets using a `ConditionSet` object:

$sol_f \cup \{x | x \in \mathbb{R} \wedge g = 0\}$, where sol_f is the solution of the equation $f = 0$.

- Using a dict may lead to surprising results like:
 - `solve(Eq(x**2, 1), x) != solve(Eq(y**2, 1), y)`

Mathematically, this doesn't make sense. Using `FiniteSet` here solves the problem.
- It also cannot represent solutions for equations like $|x| < 1$, which is a disk of radius 1 in the Argand Plane. This problem is solved using complex sets implemented as `ComplexRegion`.

5.34.4 Input API of solveset

`solveset` has a cleaner input API, unlike `solve`. It takes a maximum of three arguments:

`solveset(equation, variable=None, domain=S.Complexes)`

- Equation(s)
The equation(s) to solve.
- Variable(s)
The variable(s) for which the equation is to be solved.
- Domain
The domain in which the equation is to be solved.

`solveset` removes the `flags` argument of `solve`, which had made the input API messy and output API inconsistent.

5.34.5 What is this domain argument about?

`Solveset` is designed to be independent of the assumptions on the variable being solved for and instead, uses the `domain` argument to decide the solver to dispatch the equation to, namely `solveset_real` or `solveset_complex`. It's unlike the old `solve` which considers the assumption on the variable.

```
>>> from sympy import solveset, S
>>> from sympy.abc import x
>>> solveset(x**2 + 1, x) # domain=S.Complexes is default
{-I, I}
>>> solveset(x**2 + 1, x, domain=S.Reals)
EmptySet()
```

5.34.6 What are the general methods employed by solveset to solve an equation?

Solveset uses various methods to solve an equation, here is a brief overview of the methodology:

- The domain argument is first considered to know the domain in which the user is interested to get the solution.
- If the given function is a relational ($\geq$, $\leq$, $>$, $<$), and the domain is real, then `solve_univariate_inequality` and solutions are returned. Solving for complex solutions of inequalities, like $x^2 < 0$ is not yet supported.
- Based on the domain, the equation is dispatched to one of the two functions `solveset_real` or `solveset_complex`, which solves the given equation in the complex or real domain, respectively.
- If the given expression is a product of two or more functions, like say $gh = 0$, then the solution to the given equation is the Union of the solution of the equations $g = 0$ and $h = 0$, if and only if both g and h are finite for a finite input. So, the solution is built up recursively.
- If the function is trigonometric or hyperbolic, the function `_solve_real_trig` is called, which solves it by converting it to complex exponential form.
- The function is now checked if there is any instance of a `Piecewise` expression, if it is, then it's converted to explicit expression and set pairs and then solved recursively.
- The respective solver now tries to invert the equation using the routines `invert_real` and `invert_complex`. These routines are based on the concept of mathematical inverse (though not exactly). It reduces the real/complex valued equation $f(x) = y$ to a set of equations: $\{g(x) = h_1(y), g(x) = h_2(y), \dots, g(x) = h_n(y)\}$ where $g(x)$ is a simpler function than $f(x)$. There is some work needed to be done in this to find invert of more complex expressions.
- After the invert, the equations are checked for radical or Abs (Modulus), then the method `_solve_radical` tries to simplify the radical, by removing it using techniques like squaring, cubing etc, and `_solve_abs` solves nested Modulus by considering the positive and negative variants, iteratively.
- If none of the above method is successful, then methods of polynomial is used as follows:
 - The method to solve the rational function, `_solve_as_rational`, is called. Based on the domain, the respective poly solver `_solve_as_poly_real` or `_solve_as_poly_complex` is called to solve f as a polynomial.
 - The underlying method `_solve_as_poly` solves the equation using polynomial techniques if it's already a polynomial equation or, with a change of variables, can be made so.

- The final solution set returned by `solveset` is the intersection of the set of solutions found above and the input domain.

5.34.7 How do we manipulate and return an infinite solution?

- In the real domain, we use our `ImageSet` class in the `sets` module to return infinite solutions. `ImageSet` is an image of a set under a mathematical function. For example, to represent the solution of the equation $\sin(x) = 0$, we can use the `ImageSet` as:

```
>>> from sympy import ImageSet, Lambda, pi, S, Dummy, pprint
>>> n = Dummy('n')
>>> pprint(ImageSet(Lambda(n, 2*pi*n), S.Integers), use_unicode=True)
{2·n·π | n ∈ ℤ}
```

Where n is a dummy variable. It is basically the image of the set of integers under the function $2\pi n$.

- In the complex domain, we use complex sets, which are implemented as the `ComplexRegion` class in the `sets` module, to represent infinite solution in the Argand plane. For example to represent the solution of the equation $|z| = 1$, which is a unit circle, we can use the `ComplexRegion` as:

```
>>> from sympy import ComplexRegion, FiniteSet, Interval, pi, pprint
>>> pprint(ComplexRegion(FiniteSet(1)*Interval(0, 2*pi), polar=True), use_
→unicode=True)
{r·(i·sin(θ) + cos(θ)) | r, θ ∈ {1} × [0, 2·π]}
```

Where the `FiniteSet` in the `ProductSet` is the range of the value of r , which is the radius of the circle and the `Interval` is the range of θ , the angle from the x axis representing a unit circle in the Argand plane.

Note: We also have non-polar form notation for representing solution in rectangular form. For example, to represent first two quadrants in the Argand plane, we can write the `ComplexRegion` as:

```
>>> from sympy import ComplexRegion, Interval, pi, oo, pprint
>>> pprint(ComplexRegion(Interval(-oo, oo)*Interval(0, oo)), use_unicode=True)
{x + y·i | x, y ∈ (-∞, ∞) × [0, ∞)}
```

where the `Intervals` are the range of x and y for the set of complex numbers $x + iy$.

5.34.8 How does `solveset` ensure that it is not returning any wrong solution?

Solvers in a Computer Algebra System are based on heuristic algorithms, so it's usually very hard to ensure 100% percent correctness, in every possible case. However there are still a lot of cases where we can ensure correctness. `Solveset` tries to verify correctness wherever it can. For example:

Consider the equation $|x| = n$. A naive method to solve this equation would return $\{-n, n\}$ as its solution, which is not correct since $\{-n, n\}$ can be its solution if and only if n is positive. `Solveset` returns this information as well to ensure correctness.

```
>>> from sympy import symbols, S, pprint, solveset
>>> x, n = symbols('x, n')
>>> pprint(solveset(abs(x) - n, x, domain=S.Reals), use_unicode=True)
{x | x ∈ {-n, n} ∧ (n ∈ [0, ∞))}
```

Though, there still a lot of work needs to be done in this regard.

5.34.9 Search based solver and step-by-step solution

Note: This is under Development.

After the introduction of ConditionSet, the solving of equations can be seen as set transformations. Here is an abstract view of the things we can do to solve equations.

- Apply various set transformations on the given set.
- Define a metric of the usability of solutions, or a notion of some solutions being better than others.
- Different transformations would be the nodes of a tree.
- Suitable searching techniques could be applied to get the best solution.

ConditionSet gives us the ability to represent unevaluated equations and inequalities in forms like $\{x|f(x) = 0; x \in S\}$ and $\{x|f(x) > 0; x \in S\}$ but a more powerful thing about ConditionSet is that it allows us to write the intermediate steps as set to set transformation. Some of the transformations are:

- **Composition:** $\{x|f(g(x)) = 0; x \in S\} \Rightarrow \{x|g(x) = y; x \in S, y \in \{z|f(z) = 0; z \in S\}\}$
- **Polynomial Solver:** $\{x|P(x) = 0; x \in S\} \Rightarrow \{x_1, x_2, \dots, x_n\} \cap S$, where x_i are roots of $P(x)$.
- **Invert solver:** $\{x|f(x) = 0; x \in S\} \Rightarrow \{g(0)| \text{ all } g \text{ such that } f(g(x)) = x\}$
- **logcombine:** $\{x|\log(f(x)) + \log(g(x)); x \in S\} \Rightarrow \{x|\log(f(x).g(x)); x \in S\}$ if $f(x) > 0$ and $g(x) > 0 \Rightarrow \{x|\log(f(x)) + \log(g(x)); x \in S\}$ otherwise
- **product solve:** $\{x|f(x)g(x) = 0; x \in S\} \Rightarrow \{x|f(x) = 0; x \in S\} \cup \{x|g(x) = 0; x \in S\}$ given $f(x)$ and $g(x)$ are bounded. $\Rightarrow \{x|f(x)g(x) = 0; x \in S\}$, otherwise

Since the output type is same as the input type any composition of these transformations is also a valid transformation. And our aim is to find the right sequence of compositions (given the atoms) which transforms the given condition set to a set which is not a condition set i.e., FiniteSet, Interval, Set of Integers and their Union, Intersection, Complement or ImageSet. We can assign a cost function to each set, such that, the more desirable that form of set is to us, the less the value of the cost function. This way our problem is now reduced to finding the path from the initial ConditionSet to the lowest valued set on a graph where the atomic transformations forms the edges.

5.34.10 How do we deal with cases where only some of the solutions are known?

Creating a universal equation solver, which can solve each and every equation we encounter in mathematics is an ideal case for solvers in a Computer Algebra System. When cases which are not solved or can only be solved incompletely, a ConditionSet is used and acts as an unevaluated solveset object.

Note that, mathematically, finding a complete set of solutions for an equation is undecidable. See [Richardson's theorem](#).

ConditionSet is basically a Set of elements which satisfy a given condition. For example, to represent the solutions of the equation in the real domain:

$$(x^2 - 4)(\sin(x) + x)$$

We can represent it as:

$$\{-2, 2\} \cup \{x | x \in \mathbb{R} \wedge x + \sin(x) = 0\}$$

5.34.11 What will you do with the old solve?

There are still a few things solveset can't do, which the old solve can, such as solving non linear multivariate & LambertW type equations. Hence, it's not yet a perfect replacement for old solve. The ultimate goal is to:

- Replace solve with solveset once solveset is at least as powerful as solve, i.e., solveset does everything that solve can do currently, and
- eventually rename solveset to solve.

5.34.12 How are symbolic parameters handled in solveset?

Solveset is in its initial phase of development, so the symbolic parameters aren't handled well for all the cases, but some work has been done in this regard to depict our ideology towards symbolic parameters. As an example, consider the solving of $|x| = n$ for real x , where n is a symbolic parameter. Solveset returns the value of x considering the domain of the symbolic parameter n as well:

$$([0, \infty) \cap \{n\}) \cup ((-\infty, 0] \cap \{-n\}).$$

This simply means n is the solution only when it belongs to the Interval $[0, \infty)$ and $-n$ is the solution only when $-n$ belongs to the Interval $(-\infty, 0]$.

There are other cases to address too, like solving $2^x + (a - 2)$ for x where a is a symbolic parameter. As of now, It returns the solution as an intersection with $\mathbb{R}$, which is trivial, as it doesn't reveal the domain of a in the solution.

Recently, we have also implemented a function to find the domain of the expression in a FiniteSet (Intersection with the interval) in which it is not-empty. It is a useful addition for dealing with symbolic parameters. For example:

```
>>> from sympy import Symbol, FiniteSet, Interval, not_empty_in, sqrt, oo
>>> from sympy.abc import x
>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)
>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(-sqrt(2), -1), Interval(1, 2))
```

5.34.13 References

5.34.14 Solveset Module Reference

Use `solveset()` (page 1350) to solve equations or expressions (assumed to be equal to 0) for a single variable. Solving an equation like $x^2 == 1$ can be done as follows:

```
>>> from sympy import solveset
>>> from sympy import Symbol, Eq
>>> x = Symbol('x')
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
```

Or one may manually rewrite the equation as an expression equal to 0:

```
>>> solveset(x**2 - 1, x)
{-1, 1}
```

The first argument for `solveset()` (page 1350) is an expression (equal to zero) or an equation and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solveset.solveset(f, symbol=None, domain=S.Complexes)`

Solves a given inequality or equation with set as output

Parameters *f* : Expr or a relational.

The target equation or inequality

symbol : Symbol

The variable for which the equation is solved

domain : Set

The domain over which the equation is solved

Returns Set

A set of values for *symbol* for which *f* is True or is equal to zero. An *EmptySet* is returned if *f* is False or nonzero. A *ConditionSet* is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

solveset claims to be complete in the solution set that it returns.

Raises `NotImplementedError`

The algorithms to solve inequalities in complex domain are not yet implemented.

ValueError

The input is not valid.

RuntimeError

It is a bug, please report to the github issue tracker.

See also:

`solveset_real` (page 1351) solver for real domain

`solveset_complex` (page 1351) solver for complex domain

Notes

Python interprets 0 and 1 as False and True, respectively, but in this function they refer to solutions of an expression. So 0 and 1 return the Domain and EmptySet, respectively, while True and False return the opposite (as they are assumed to be solutions of relational expressions).

Examples

```
>>> from sympy import exp, sin, Symbol, pprint, S
>>> from sympy.solvers.solveset import solveset, solveset_real
```

- The default domain is complex. Not specifying a domain will lead to the solving of the equation in the complex domain (and this is not affected by the assumptions on the symbol):

```
>>> x = Symbol('x')
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

```
>>> x = Symbol('x', real=True)
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

- If you want to use *solveset* to solve the equation in the real domain, provide a real domain. (Using *solveset_real* does this automatically.)

```
>>> R = S.Reals
>>> x = Symbol('x')
>>> solveset(exp(x) - 1, x, R)
{0}
>>> solveset_real(exp(x) - 1, x)
{0}
```

The solution is mostly unaffected by assumptions on the symbol, but there may be some slight difference:

```
>>> pprint(solveset(sin(x)/x, x), use_unicode=False)
({2*n*pi | n in Integers} \ {0}) U ({2*n*pi + pi | n in Integers} \ {0})
```

```
>>> p = Symbol('p', positive=True)
>>> pprint(solveset(sin(p)/p, p), use_unicode=False)
{2*n*pi | n in Integers} U {2*n*pi + pi | n in Integers}
```

- Inequalities can be solved over the real domain only. Use of a complex domain leads to a `NotImplementedError`.

```
>>> solveset(exp(x) > 1, x, R)
Interval.open(0, oo)
```

```
sympy.solvers.solveset.solveset_real(f, symbol)
```

```
sympy.solvers.solveset.solveset_complex(f, symbol)
```

`sympy.solvers.solveset.invert_real(f_x, y, x, domain=Reals)`

Inverts a real-valued function. Same as `_invert`, but sets the domain to `S.Reals` before inverting.

`sympy.solvers.solveset.invert_complex(f_x, y, x, domain=S.Complexes)`

Reduce the complex valued equation $f(x) = y$ to a set of equations $\{g(x) = h_1(y), g(x) = h_2(y), \dots, g(x) = h_n(y)\}$ where $g(x)$ is a simpler function than $f(x)$. The return value is a tuple $(g(x), \text{set}_h)$, where $g(x)$ is a function of x and set_h is the set of function $\{h_1(y), h_2(y), \dots, h_n(y)\}$. Here, y is not necessarily a symbol.

The `set_h` contains the functions, along with the information about the domain in which they are valid, through set operations. For instance, if $y = \text{Abs}(x) - n$ is inverted in the real domain, then `set_h` is not simply $-n, n$ as the nature of n is unknown; rather, it is: $\text{Intersection}([0, \infty)n) \cup \text{Intersection}((-\infty, 0], -n)$

By default, the complex domain is used which means that inverting even seemingly simple functions like $\exp(x)$ will give very different results from those obtained in the real domain. (In the case of $\exp(x)$, the inversion via $\log$ is multi-valued in the complex domain, having infinitely many branches.)

If you are working with real values only (or you are not sure which function to use) you should probably set the domain to `S.Reals` (or use `invert_real` which does that automatically).

See also:

[invert_real](#) (page 1352), [invert_complex](#) (page 1352)

Examples

```
>>> from sympy.solvers.solveset import invert_complex, invert_real
>>> from sympy.abc import x, y
>>> from sympy import exp, log
```

When does $\exp(x) == y$?

```
>>> invert_complex(exp(x), y, x)
(x, ImageSet(Lambda(_n, I*(2*_n*pi + arg(y)) + log(Abs(y))), Integers))
>>> invert_real(exp(x), y, x)
(x, Intersection(Reals, {log(y)}))
```

When does $\exp(x) == 1$?

```
>>> invert_complex(exp(x), 1, x)
(x, ImageSet(Lambda(_n, 2*_n*I*pi), Integers))
>>> invert_real(exp(x), 1, x)
(x, {0})
```

`sympy.solvers.solveset.domain_check(f, symbol, p)`

Returns False if point p is infinite or any subexpression of f is infinite or becomes so after replacing symbol with p . If none of these conditions is met then True will be returned.

Examples

```
>>> from sympy import Mul, oo
>>> from sympy.abc import x
>>> from sympy.solvers.solveset import domain_check
>>> g = 1/(1 + (1/(x + 1))**2)
>>> domain_check(g, x, -1)
False
>>> domain_check(x**2, x, 0)
True
>>> domain_check(1/x, x, oo)
False
```

- The function relies on the assumption that the original form of the equation has not been changed by automatic simplification.

```
>>> domain_check(x/x, x, 0) # x/x is automatically simplified to 1
True
```

- To deal with automatic evaluations use `evaluate=False`:

```
>>> domain_check(Mul(x, 1/x, evaluate=False), x, 0)
False
```

5.34.15 linear_eq_to_matrix

`sympy.solvers.solveset.linear_eq_to_matrix(equations, *symbols)`

Converts a given System of Equations into Matrix form. Here *equations* must be a linear system of equations in *symbols*. The order of symbols in input *symbols* will determine the order of coefficients in the returned Matrix.

The Matrix form corresponds to the augmented matrix form. For example:

$$4x + 2y + 3z = 1$$

$$3x + y + z = -6$$

$$2x + 4y + 9z = 2$$

This system would return *A* & *b* as given below:

$$A = \begin{bmatrix} 4 & 2 & 3 \\ 3 & 1 & 1 \\ 2 & 4 & 9 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ -6 \\ 2 \end{bmatrix}$$

Examples

```
>>> from sympy import linear_eq_to_matrix, symbols
>>> x, y, z = symbols('x, y, z')
>>> eqns = [x + 2*y + 3*z - 1, 3*x + y + z + 6, 2*x + 4*y + 9*z - 2]
>>> A, b = linear_eq_to_matrix(eqns, [x, y, z])
```

(continues on next page)

(continued from previous page)

```

>>> A
Matrix([
[1, 2, 3],
[3, 1, 1],
[2, 4, 9]])
>>> b
Matrix([
[ 1],
[-6],
[ 2]])
>>> eqns = [x + z - 1, y + z, x - y]
>>> A, b = linear_eq_to_matrix(eqns, [x, y, z])
>>> A
Matrix([
[1, 0, 1],
[0, 1, 1],
[1, -1, 0]])
>>> b
Matrix([
[1],
[0],
[0]])

```

- Symbolic coefficients are also supported

```

>>> a, b, c, d, e, f = symbols('a, b, c, d, e, f')
>>> eqns = [a*x + b*y - c, d*x + e*y - f]
>>> A, B = linear_eq_to_matrix(eqns, x, y)
>>> A
Matrix([
[a, b],
[d, e]])
>>> B
Matrix([
[c],
[f]])

```

5.34.16 linsolve

`sympy.solvers.solve.set.linsolve(system, *symbols)`

Solve system of N linear equations with M variables, which means both under - and overdetermined systems are supported. The possible number of solutions is zero, one or infinite. Zero solutions throws a `ValueError`, where as infinite solutions are represented parametrically in terms of given symbols. For unique solution a `FiniteSet` of ordered tuple is returned.

All Standard input formats are supported: For the given set of Equations, the respective input types are given below:

$$3x + 2y - z = 1$$

$$2x - 2y + 4z = -2$$

$$2x - y + 2z = 0$$

- Augmented Matrix Form, *system* given below:

```
system =  $\begin{bmatrix} 3 & 2 & -1 & 1 \\ 2 & -2 & 4 & -2 \\ 2 & -1 & 2 & 0 \end{bmatrix}$ 
```

- List Of Equations Form

```
system =  $[3x + 2y - z - 1, 2x - 2y + 4z + 2, 2x - y + 2z]$ 
```

- Input A & b Matrix Form (from $Ax = b$) are given as below:

```
A =  $\begin{bmatrix} 3 & 2 & -1 \\ 2 & -2 & 4 \\ 2 & -1 & 2 \end{bmatrix}$       b =  $\begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$ 
```

```
system = (A, b)
```

Symbols can always be passed but are actually only needed when 1) a system of equations is being passed and 2) the system is passed as an underdetermined matrix and one wants to control the name of the free variables in the result. An error is raised if no symbols are used for case 1, but if no symbols are provided for case 2, internally generated symbols will be provided. When providing symbols for case 2, there should be at least as many symbols as there are columns in matrix A.

The algorithm used here is Gauss-Jordan elimination, which results, after elimination, in a row echelon form matrix.

Returns A FiniteSet containing an ordered tuple of values for the unknowns for which the *system* has a solution. (Wrapping the tuple in FiniteSet is used to maintain a consistent output format throughout solveset.)

Returns EmptySet(), if the linear system is inconsistent.

Raises ValueError

The input is not valid. The symbols are not given.

Examples

```
>>> from sympy import Matrix, S, linsolve, symbols
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> A
Matrix([
[1, 2, 3],
[4, 5, 6],
[7, 8, 10]])
>>> b
Matrix([
[3],
[6],
[9]])
>>> linsolve((A, b), [x, y, z])
{(-1, 2, 0)}
```

- Parametric Solution: In case the system is underdetermined, the function will return a parametric solution in terms of the given symbols. Those that are free will be returned unchanged. e.g. in the system below, z is returned as the solution for variable z ; it can take on any value.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), x, y, z)
{(z - 1, -2*z + 2, z)}
```

If no symbols are given, internally generated symbols will be used. The *tau0* in the 3rd position indicates (as before) that the 3rd variable – whatever it's named – can take on any value:

```
>>> linsolve((A, b))
{(tau0 - 1, -2*tau0 + 2, tau0)}
```

- List of Equations as input

```
>>> Eqns = [3*x + 2*y - z - 1, 2*x - 2*y + 4*z + 2, -x + y/2 - z]
>>> linsolve(Eqns, x, y, z)
{(1, -2, -2)}
```

- Augmented Matrix as input

```
>>> aug = Matrix([[2, 1, 3, 1], [2, 6, 8, 3], [6, 8, 18, 5]])
>>> aug
Matrix([
[2, 1, 3, 1],
[2, 6, 8, 3],
[6, 8, 18, 5]])
>>> linsolve(aug, x, y, z)
{(3/10, 2/5, 0)}
```

- Solve for symbolic coefficients

```
>>> a, b, c, d, e, f = symbols('a, b, c, d, e, f')
>>> eqns = [a*x + b*y - c, d*x + e*y - f]
>>> linsolve(eqns, x, y)
{((-b*f + c*e)/(a*e - b*d), (a*f - c*d)/(a*e - b*d))}
```

- A degenerate system returns solution as set of given symbols.

```
>>> system = Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
>>> linsolve(system, x, y)
{(x, y)}
```

- For an empty system linsolve returns empty set

```
>>> linsolve([], x)
EmptySet()
```

5.34.17 nonlinsolve

`sympy.solvers.solve.set.nonlinsolve(system, *symbols)`

Solve system of N non linear equations with M variables, which means both under and overdetermined systems are supported. Positive dimensional system is also supported (A system with infinitely many solutions is said to be positive-dimensional). In Positive dimensional system solution will be dependent on at least one symbol. Returns both real solution and complex solution(If system have). The possible number of solutions is zero, one or infinite.

Parameters **system** : list of equations

The target system of equations

symbols : list of Symbols

symbols should be given as a sequence eg. list

Returns A FiniteSet of ordered tuple of values of *symbols* for which the *system* has solution. Order of values in the tuple is same as symbols present in the parameter *symbols*.

Please note that general FiniteSet is unordered, the solution returned here is not simply a FiniteSet of solutions, rather it is a FiniteSet of ordered tuple, i.e. the first & only argument to FiniteSet is a tuple of solutions, which is ordered, & hence the returned solution is ordered.

Also note that solution could also have been returned as an ordered tuple, FiniteSet is just a wrapper around the tuple. It has no other significance except for the fact it is just used to maintain a consistent output format throughout the solveset.

For the given set of Equations, the respective input types are given below:

$$x * y - 1 = 0$$

$$4 * x ** 2 + y ** 2 - 5 = 0$$

system = $[x * y - 1, 4 * x ** 2 + y ** 2 - 5]$

symbols = $[x, y]$

Raises ValueError

The input is not valid. The symbols are not given.

AttributeError

The input symbols are not *Symbol* type.

Examples

```
>>> from sympy.core.symbol import symbols
>>> from sympy.solvers.solveset import nonlinsolve
>>> x, y, z = symbols('x, y, z', real=True)
>>> nonlinsolve([x*y - 1, 4*x**2 + y**2 - 5], [x, y])
{(-1, -1), (-1/2, -2), (1/2, 2), (1, 1)}
```

1. Positive dimensional system and complements:

```
>>> from sympy import pprint
>>> from sympy.polys.polytools import is_zero_dimensional
>>> a, b, c, d = symbols('a, b, c, d', real=True)
>>> eq1 = a + b + c + d
>>> eq2 = a*b + b*c + c*d + d*a
>>> eq3 = a*b*c + b*c*d + c*d*a + d*a*b
>>> eq4 = a*b*c*d - 1
>>> system = [eq1, eq2, eq3, eq4]
>>> is_zero_dimensional(system)
False
>>> pprint(nonlinsolve(system, [a, b, c, d]), use_unicode=False)
-1      1      1      -1
{(- --, -d, -, {d} \ {0}), (-, -d, ---, {d} \ {0})}
  d      d      d      d
>>> nonlinsolve([(x+y)**2 - 4, x + y - 2], [x, y])
{(-y + 2, y)}
```

2. If some of the equations are non polynomial equation then *nonlinsolve* will call *substitution* function and returns real and complex solutions, if present.

```
>>> from sympy import exp, sin
>>> nonlinsolve([exp(x) - sin(y), y**2 - 4], [x, y])
{(log(sin(2)), 2), (ImageSet(Lambda(_n, I*(2*_n*pi + pi) +
    log(sin(2))), Integers), -2), (ImageSet(Lambda(_n, 2*_n*I*pi +
    Mod(log(sin(2)), 2*I*pi)), Integers), 2)}
```

3. If system is Non linear polynomial zero dimensional then it returns both solution (real and complex solutions, if present using *solve_poly_system*):

```
>>> from sympy import sqrt
>>> nonlinsolve([x**2 - 2*y**2 - 2, x*y - 2], [x, y])
{(-2, -1), (2, 1), (-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I)}
```

4. *nonlinsolve* can solve some linear(zero or positive dimensional) system (because it is using *groebner* function to get the groebner basis and then *substitution* function basis as the new *system*). But it is not recommended to solve linear system using *nonlinsolve*, because *linsolve* is better for all kind of linear system.

```
>>> nonlinsolve([x + 2*y - z - 3, x - y - 4*z + 9, y + z - 4], [x, y, z])
{(3*z - 5, -z + 4, z)}
```

5. System having polynomial equations and only real solution is present (will be solved using *solve_poly_system*):

```
>>> e1 = sqrt(x**2 + y**2) - 10
>>> e2 = sqrt(y**2 + (-x + 10)**2) - 3
>>> nonlinsolve((e1, e2), (x, y))
{(191/20, -3*sqrt(391)/20), (191/20, 3*sqrt(391)/20)}
```

(continues on next page)

(continued from previous page)

```
>>> nonlinsolve([x**2 + 2/y - 2, x + y - 3], [x, y])
{(1, 2), (1 + sqrt(5), -sqrt(5) + 2), (-sqrt(5) + 1, 2 + sqrt(5))}
>>> nonlinsolve([x**2 + 2/y - 2, x + y - 3], [y, x])
{(2, 1), (2 + sqrt(5), -sqrt(5) + 1), (-sqrt(5) + 2, 1 + sqrt(5))}
```

6. It is better to use symbols instead of Trigonometric Function or Function (e.g. replace $\sin(x)$ with symbol, replace $f(x)$ with symbol and so on. Get soln from *nonlinsolve* and then using *solveset* get the value of x)

How Nonlinsolve Is Better Than Old Solver *solve_system* :

1. A positive dimensional system solver : *nonlinsolve* can return solution for positive dimensional system. It finds the Groebner Basis of the positive dimensional system (calling it as basis) then we can start solving equation (having least number of variable first in the basis) using *solveset* and substituting that solved solutions into other equation (of basis) to get solution in terms of minimum variables. Here the important thing is how we are substituting the known values and in which equations.

2. Real and Complex both solutions : *nonlinsolve* returns both real and complex solution. If all the equations in the system are polynomial then using *solve_poly_system* both real and complex solution is returned. If all the equations in the system are not polynomial equation(s) then goes to *substitution* method with this polynomial and non polynomial equation(s), to solve for unsolved variables. Here to solve for particular variable *solveset_real* and *solveset_complex* is used. For both real and complex solution function *solve_using_known_values* is used inside *substitution* function. (*substitution* function will be called when there is any non polynomial equation(s) is present). When solution is valid then add its general solution in the final result.

3. Complement and Intersection will be added if any : *nonlinsolve* maintains dict for complements and Intersections. If *solveset* find complements or/and Intersection with any Interval or set during the execution of *substitution* function, then complement or/and Intersection for that variable is added before returning final solution.

5.34.18 Diophantine Equations (DEs)

See [Diophantine](#) (page 1314)

5.34.19 Inequalities

See [Inequality Solvers](#) (page 1339)

5.34.20 Ordinary Differential equations (ODEs)

See [ODE](#) (page 1230).

5.34.21 Partial Differential Equations (PDEs)

See [PDE](#) (page 1287).

5.35 Tensor Module

A module to manipulate symbolic objects with indices including tensors

5.35.1 Contents

N-dim array

N-dim array module for SymPy.

Four classes are provided to handle N-dim arrays, given by the combinations dense/sparse (i.e. whether to store all elements or only the non-zero ones in memory) and mutable/immutable (immutable classes are SymPy objects, but cannot change after they have been created).

Examples

The following examples show the usage of Array. This is an abbreviation for ImmutableDenseNDimArray, that is an immutable and dense N-dim array, the other classes are analogous. For mutable classes it is also possible to change element values after the object has been constructed.

Array construction can detect the shape of nested lists and tuples:

```
>>> from sympy import Array
>>> a1 = Array([[1, 2], [3, 4], [5, 6]])
>>> a1
[[1, 2], [3, 4], [5, 6]]
>>> a1.shape
(3, 2)
>>> a1.rank()
2
>>> from sympy.abc import x, y, z
>>> a2 = Array([[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]])
>>> a2
[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]
>>> a2.shape
(2, 2, 2)
>>> a2.rank()
3
```

Otherwise one could pass a 1-dim array followed by a shape tuple:

```
>>> m1 = Array(range(12), (3, 4))
>>> m1
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
>>> m2 = Array(range(12), (3, 2, 2))
>>> m2
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> m2[1,1,1]
7
>>> m2.reshape(4, 3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

Slice support:

```
>>> m2[:, 1, 1]
[3, 7, 11]
```

Elementwise derivative:

```
>>> from sympy.abc import x, y, z
>>> m3 = Array([x**3, x*y, z])
>>> m3.diff(x)
[3*x**2, y, 0]
>>> m3.diff(z)
[0, 0, 1]
```

Multiplication with other SymPy expressions is applied elementwisely:

```
>>> (1+x)*m3
[x**3*(x + 1), x*y*(x + 1), z*(x + 1)]
```

To apply a function to each element of the N-dim array, use `applyfunc`:

```
>>> m3.applyfunc(lambda x: x/2)
[x**3/2, x*y/2, z/2]
```

N-dim arrays can be converted to nested lists by the `tolist()` method:

```
>>> m2.tolist()
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> isinstance(m2.tolist(), list)
True
```

If the rank is 2, it is possible to convert them to matrices with `tomatrix()`:

```
>>> m1.tomatrix()
Matrix([
[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]])
```

Products and contractions

Tensor product between arrays $A_{i_1, \dots, i_n}$ and $B_{j_1, \dots, j_m}$ creates the combined array $P = A \otimes B$ defined as

$$P_{i_1, \dots, i_n, j_1, \dots, j_m} := A_{i_1, \dots, i_n} \cdot B_{j_1, \dots, j_m}.$$

It is available through `tensorproduct(...)`:

```
>>> from sympy import Array, tensorproduct
>>> from sympy.abc import x, y, z, t
>>> A = Array([x, y, z, t])
>>> B = Array([1, 2, 3, 4])
>>> tensorproduct(A, B)
[[x, 2*x, 3*x, 4*x], [y, 2*y, 3*y, 4*y], [z, 2*z, 3*z, 4*z], [t, 2*t, 3*t, 4*t]]
```

Tensor product between a rank-1 array and a matrix creates a rank-3 array:

```
>>> from sympy import eye
>>> p1 = tensorproduct(A, eye(4))
>>> p1
[[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]], [[y, 0, 0, 0], [0, y, 0, 0],
↪ [0, 0, y, 0], [0, 0, 0, y]], [[z, 0, 0, 0], [0, z, 0, 0], [0, 0, z, 0], [0, 0, 0, z]],
↪ [[t, 0, 0, 0], [0, t, 0, 0], [0, 0, t, 0], [0, 0, 0, t]]]
```

Now, to get back $A_0 \otimes \mathbf{1}$ one can access $p_{0,m,n}$ by slicing:

```
>>> p1[0,:,:]
[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]]
```

Tensor contraction sums over the specified axes, for example contracting positions a and b means

$$A_{i_1, \dots, i_a, \dots, i_b, \dots, i_n} \Rightarrow \sum_k A_{i_1, \dots, k, \dots, k, \dots, i_n}$$

Remember that Python indexing is zero starting, to contract the a -th and b -th axes it is therefore necessary to specify $a - 1$ and $b - 1$

```
>>> from sympy import tensorcontraction
>>> C = Array([[x, y], [z, t]])
```

The matrix trace is equivalent to the contraction of a rank-2 array:

$$A_{m,n} \Rightarrow \sum_k A_{k,k}$$

```
>>> tensorcontraction(C, (0, 1))
t + x
```

Matrix product is equivalent to a tensor product of two rank-2 arrays, followed by a contraction of the 2nd and 3rd axes (in Python indexing axes number 1, 2).

$$A_{m,n} \cdot B_{i,j} \Rightarrow \sum_k A_{m,k} \cdot B_{k,j}$$

```
>>> D = Array([[2, 1], [0, -1]])
>>> tensorcontraction(tensorproduct(C, D), (1, 2))
[[2*x, x - y], [2*z, -t + z]]
```

One may verify that the matrix product is equivalent:

```
>>> from sympy import Matrix
>>> Matrix([[x, y], [z, t]])*Matrix([[2, 1], [0, -1]])
Matrix([
  2*x,  x - y],
  2*z, -t + z]])
```

or equivalently

```
>>> C.tomatrix()*D.tomatrix()
Matrix([
  2*x,  x - y],
  2*z, -t + z]])
```

Derivatives by array

The usual derivative operation may be extended to support derivation with respect to arrays, provided that all elements in the that array are symbols or expressions suitable for derivations.

The definition of a derivative by an array is as follows: given the array $A_{i_1, \dots, i_N}$ and the array $X_{j_1, \dots, j_M}$ the derivative of arrays will return a new array B defined by

$$B_{j_1, \dots, j_M, i_1, \dots, i_N} := \frac{\partial A_{i_1, \dots, i_N}}{\partial X_{j_1, \dots, j_M}}$$

The function `derive_by_array` performs such an operation:

```
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> from sympy import sin, exp
```

With scalars, it behaves exactly as the ordinary derivative:

```
>>> derive_by_array(sin(x*y), x)
y*cos(x*y)
```

Scalar derived by an array basis:

```
>>> derive_by_array(sin(x*y), [x, y, z])
[y*cos(x*y), x*cos(x*y), 0]
```

Deriving array by an array basis: $B^{nm} := \frac{\partial A^m}{\partial x^n}$

```
>>> basis = [x, y, z]
>>> ax = derive_by_array([exp(x), sin(y*z), t], basis)
>>> ax
[[exp(x), 0, 0], [0, z*cos(y*z), 0], [0, y*cos(y*z), 0]]
```

Contraction of the resulting array: $\sum_m \frac{\partial A^m}{\partial x^m}$

```
>>> tensorcontraction(ax, (0, 1))
z*cos(y*z) + exp(x)
```

Classes

```
class sympy.tensor.array.ImmutableDenseNDimArray
```

```
class sympy.tensor.array.ImmutableSparseNDimArray
```

```
class sympy.tensor.array.MutableDenseNDimArray
```

```
class sympy.tensor.array.MutableSparseNDimArray
```

Functions

```
sympy.tensor.array.derive_by_array(expr, dx)
```

Derivative by arrays. Supports both arrays and scalars.

Given the array $A_{i_1, \dots, i_N}$ and the array $X_{j_1, \dots, j_M}$ this function will return a new array B defined by

$$B_{j_1, \dots, j_M, i_1, \dots, i_N} := \frac{\partial A_{i_1, \dots, i_N}}{\partial X_{j_1, \dots, j_M}}$$

Examples

```
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> from sympy import cos
>>> derive_by_array(cos(x*t), x)
-t*sin(t*x)
>>> derive_by_array(cos(x*t), [x, y, z, t])
[-t*sin(t*x), 0, 0, -x*sin(t*x)]
>>> derive_by_array([x, y**2*z], [[x, y], [z, t]])
[[[1, 0], [0, 2*y*z]], [[0, y**2], [0, 0]]]
```

`sympy.tensor.array.permutedims(expr, perm)`
Permutes the indices of an array.
Parameter specifies the permutation of the indices.

Examples

```
>>> from sympy.abc import x, y, z, t
>>> from sympy import sin
>>> from sympy import Array, permutedims
>>> a = Array([x, y, z], [t, sin(x), 0])
>>> a
[[x, y, z], [t, sin(x), 0]]
>>> permutedims(a, (1, 0))
[[x, t], [y, sin(x)], [z, 0]]
```

If the array is of second order, transpose can be used:

```
>>> from sympy import transpose
>>> transpose(a)
[[x, t], [y, sin(x)], [z, 0]]
```

Examples on higher dimensions:

```
>>> b = Array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> permutedims(b, (2, 1, 0))
[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
>>> permutedims(b, (1, 2, 0))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

Permutation objects are also allowed:

```
>>> from sympy.combinatorics import Permutation
>>> permutedims(b, Permutation([1, 2, 0]))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

`sympy.tensor.array.tensorcontraction(array, *contraction_axes)`
Contraction of an array-like object on the specified axes.

Examples

```
>>> from sympy import Array, tensorcontraction
>>> from sympy import Matrix, eye
>>> tensorcontraction(eye(3), (0, 1))
3
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15, 16, 17]]]
>>> tensorcontraction(A, (0, 2))
[21, 30]
```

Matrix multiplication may be emulated with a proper combination of `tensorcontraction` and `tensorproduct`

```
>>> from sympy import tensorproduct
>>> from sympy.abc import a,b,c,d,e,f,g,h
>>> m1 = Matrix([[a, b], [c, d]])
>>> m2 = Matrix([[e, f], [g, h]])
>>> p = tensorproduct(m1, m2)
>>> p
[[[a*e, a*f], [a*g, a*h]], [[b*e, b*f], [b*g, b*h]]], [[c*e, c*f], [c*g, c*h]],
↳ [[d*e, d*f], [d*g, d*h]]]
>>> tensorcontraction(p, (1, 2))
[a*e + b*g, a*f + b*h], [c*e + d*g, c*f + d*h]]
>>> m1*m2
Matrix([
[a*e + b*g, a*f + b*h],
[c*e + d*g, c*f + d*h]])
```

`sympy.tensor.array.tensorproduct(*args)`
Tensor product among scalars or array-like objects.

Examples

```
>>> from sympy.tensor.array import tensorproduct, Array
>>> from sympy.abc import x, y, z, t
>>> A = Array([[1, 2], [3, 4]])
>>> B = Array([x, y])
>>> tensorproduct(A, B)
[[[x, y], [2*x, 2*y]], [[3*x, 3*y], [4*x, 4*y]]]
>>> tensorproduct(A, x)
[[x, 2*x], [3*x, 4*x]]
>>> tensorproduct(A, B, B)
[[[x**2, x*y], [x*y, y**2]], [[2*x**2, 2*x*y], [2*x*y, 2*y**2]]], [[3*x**2,
↳ 3*x*y], [3*x*y, 3*y**2]], [[4*x**2, 4*x*y], [4*x*y, 4*y**2]]]
```

Applying this function on two matrices will result in a rank 4 array.

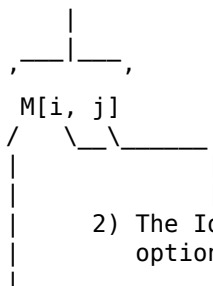
```
>>> from sympy import Matrix, eye
>>> m = Matrix([[x, y], [z, t]])
>>> p = tensorproduct(eye(3), m)
>>> p
[[[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[0, 0], [0, 0]], [[x,
↳ y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[x, y],
↳ [z, t]]]
```

Indexed Objects

Module that defines indexed objects

The classes `IndexedBase`, `Indexed`, and `Idx` represent a matrix element $M[i, j]$ as in the following diagram:

1) The `Indexed` class represents the entire indexed object.



2) The `Idx` class represents indices; each `Idx` can optionally contain information about its range.

3) `IndexedBase` represents the 'stem' of an indexed object, here 'M'. The stem used by itself is usually taken to represent the entire array.

There can be any number of indices on an `Indexed` object. No transformation properties are implemented in these Base objects, but implicit contraction of repeated indices is supported.

Note that the support for complicated (i.e. non-atomic) integer expressions as indices is limited. (This should be improved in future releases.)

Examples

=====

To express the above matrix element example you would write:

```
>>> from sympy import symbols, IndexedBase, Idx
>>> M = IndexedBase('M')
>>> i, j = symbols('i j', cls=Idx)
>>> M[i, j]
M[i, j]
```

Repeated indices in a product implies a summation, so to express a matrix-vector product in terms of `Indexed` objects:

```
>>> x = IndexedBase('x')
>>> M[i, j]*x[j]
M[i, j]*x[j]
```

If the indexed objects will be converted to component based arrays, e.g. with the code printers or the autowrap framework, you also need to provide (symbolic or numerical) dimensions. This can be done by passing an optional shape parameter to `IndexedBase` upon construction:

```
>>> dim1, dim2 = symbols('dim1 dim2', integer=True)
>>> A = IndexedBase('A', shape=(dim1, 2*dim1, dim2))
>>> A.shape
(dim1, 2*dim1, dim2)
>>> A[i, j, 3].shape
```

(continues on next page)

(continued from previous page)

```
(dim1, 2*dim1, dim2)
```

If an IndexedBase object has no shape information, it is assumed that the array is as large as the ranges of its indices:

```
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> M[i, j].shape
(m, n)
>>> M[i, j].ranges
[(0, m - 1), (0, n - 1)]
```

The above can be compared with the following:

```
>>> A[i, 2, j].shape
(dim1, 2*dim1, dim2)
>>> A[i, 2, j].ranges
[(0, m - 1), None, (0, n - 1)]
```

To analyze the structure of indexed expressions, you can use the methods `get_indices()` and `get_contraction_structure()`:

```
>>> from sympy.tensor import get_indices, get_contraction_structure
>>> get_indices(A[i, j, j])
({i}, {})
>>> get_contraction_structure(A[i, j, j])
{(j,): {A[i, j, j]}}
```

See the appropriate docstrings for a detailed explanation of the output.

class sympy.tensor.indexed.Idx

Represents an integer index as an Integer or integer expression.

There are a number of ways to create an Idx object. The constructor takes two arguments:

label An integer or a symbol that labels the index.

range Optionally you can specify a range as either

- **Symbol or integer:** This is interpreted as a dimension. Lower and upper bounds are set to 0 and `range - 1`, respectively.
- **tuple:** The two elements are interpreted as the lower and upper bounds of the range, respectively.

Note: bounds of the range are assumed to be either integer or infinite (`oo` and `-oo` are allowed to specify an unbounded range). If `n` is given as a bound, then `n.is_integer` must not return false.

For convenience, if the label is given as a string it is automatically converted to an integer symbol. (Note: this conversion is not done for range or dimension arguments.)

Examples

```
>>> from sympy import IndexedBase, Idx, symbols, oo
>>> n, i, L, U = symbols('n i L U', integer=True)
```

If a string is given for the label an integer Symbol is created and the bounds are both None:

```
>>> idx = Idx('qwerty'); idx
qwerty
>>> idx.lower, idx.upper
(None, None)
```

Both upper and lower bounds can be specified:

```
>>> idx = Idx(i, (L, U)); idx
i
>>> idx.lower, idx.upper
(L, U)
```

When only a single bound is given it is interpreted as the dimension and the lower bound defaults to 0:

```
>>> idx = Idx(i, n); idx.lower, idx.upper
(0, n - 1)
>>> idx = Idx(i, 4); idx.lower, idx.upper
(0, 3)
>>> idx = Idx(i, oo); idx.lower, idx.upper
(0, oo)
```

label

Returns the label (Integer or integer expression) of the Idx object.

Examples

```
>>> from sympy import Idx, Symbol
>>> x = Symbol('x', integer=True)
>>> Idx(x).label
x
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

lower

Returns the lower bound of the Idx.

Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

upper

Returns the upper bound of the Idx.

Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

class sympy.tensor.indexed.Indexed

Represents a mathematical object with indices.

```
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects be created via IndexedBase:

```
>>> A = IndexedBase('A')
>>> Indexed('A', i, j) == A[i, j]
True
```

base

Returns the IndexedBase of the Indexed object.

Examples

```
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

indices

Returns the indices of the Indexed object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```

ranges

Returns a list of tuples with lower and upper range of each index.

If an index does not define the data members upper and lower, the corresponding slot in the list contains None instead of a tuple.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> Indexed('A', Idx('i', 2), Idx('j', 4), Idx('k', 8)).ranges
[(0, 1), (0, 3), (0, 7)]
>>> Indexed('A', Idx('i', 3), Idx('j', 3), Idx('k', 3)).ranges
[(0, 2), (0, 2), (0, 2)]
>>> x, y, z = symbols('x y z', integer=True)
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

rank

Returns the rank of the Indexed object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

shape

Returns a list with dimensions of each index.

Dimensions is a property of the array, not of the indices. Still, if the IndexedBase does not define a shape attribute, it is assumed that the ranges of the indices correspond to the shape of the array.

```
>>> from sympy import IndexedBase, Idx, symbols
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', m)
>>> A = IndexedBase('A', shape=(n, n))
>>> B = IndexedBase('B')
>>> A[i, j].shape
(n, n)
>>> B[i, j].shape
(m, m)
```

class sympy.tensor.indexed.IndexedBase

Represent the base or stem of an indexed object

The IndexedBase class represent an array that contains elements. The main purpose of this class is to allow the convenient creation of objects of the Indexed class. The `__getitem__` method of IndexedBase returns an instance of Indexed. Alone, without indices, the IndexedBase class can be used as a notation for e.g. matrix equations, resembling what you could do with the Symbol class. But, the IndexedBase class adds functionality that is not available for Symbol instances:

- An IndexedBase object can optionally store shape information. This can be used in to check array conformance and conditions for numpy broadcasting. (TODO)

- An IndexedBase object implements syntactic sugar that allows easy symbolic representation of array operations, using implicit summation of repeated indices.
- The IndexedBase object symbolizes a mathematical structure equivalent to arrays, and is recognized as such for code generation and automatic compilation and wrapping.

```
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> A = IndexedBase('A'); A
A
>>> type(A)
<class 'sympy.tensor.indexed.IndexedBase'>
```

When an IndexedBase object receives indices, it returns an array with named axes, represented by an Indexed object:

```
>>> i, j = symbols('i j', integer=True)
>>> A[i, j, 2]
A[i, j, 2]
>>> type(A[i, j, 2])
<class 'sympy.tensor.indexed.Indexed'>
```

The IndexedBase constructor takes an optional shape argument. If given, it overrides any shape information in the indices. (But not the index ranges!)

```
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedBase('B', shape=(o, p))
>>> B[i, j].shape
(o, p)
```

label

Returns the label of the IndexedBase object.

Examples

```
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).label
A
```

offset

Returns the offset for the IndexedBase object.

This is the value added to the resulting index when the 2D Indexed object is unrolled to a 1D form. Used in code generation.

Examples

```
>>> from sympy.printing import ccode
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> l, m, n, o = symbols('l m n o', integer=True)
>>> A = IndexedBase('A', strides=(l, m, n), offset=o)
>>> i, j, k = map(Idx, 'ijk')
>>> ccode(A[i, j, k])
'A[l*i + m*j + n*k + o]'
```

shape

Returns the shape of the IndexedBase object.

Examples

```
>>> from sympy import IndexedBase, Idx, Symbol
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).shape
(x, y)
```

Note: If the shape of the IndexedBase is specified, it will override any shape information given by the indices.

```
>>> A = IndexedBase('A', shape=(x, y))
>>> B = IndexedBase('B')
>>> i = Idx('i', 2)
>>> j = Idx('j', 1)
>>> A[i, j].shape
(x, y)
>>> B[i, j].shape
(2, 1)
```

strides

Returns the strided scheme for the IndexedBase object.

Normally this is a tuple denoting the number of steps to take in the respective dimension when traversing an array. For code generation purposes `strides='C'` and `strides='F'` can also be used.

`strides='C'` would mean that code printer would unroll in row-major order and `'F'` means unroll in column major order.

Methods

Module with functions operating on IndexedBase, Indexed and Idx objects

- Check shape conformance
- Determine indices in resulting expression

etc.

Methods in this module could be implemented by calling methods on Expr objects instead. When things stabilize this could be a useful refactoring.

`sympy.tensor.index_methods.get_contraction_structure(expr)`

Determine dummy indices of `expr` and describe its structure

By dummy we mean indices that are summation indices.

The structure of the expression is determined and described as follows:

1. A conforming summation of Indexed objects is described with a dict where the keys are summation indices and the corresponding values are sets containing all terms for which the summation applies. All Add objects in the SymPy expression tree are described like this.
2. For all nodes in the SymPy expression tree that are not of type Add, the following applies:

If a node discovers contractions in one of its arguments, the node itself will be stored as a key in the dict. For that key, the corresponding value is a list of dicts, each of which is the result of a recursive call to `get_contraction_structure()`. The list contains only dicts for the non-trivial deeper contractions, omitting dicts with None as the one and only key.

Note: The presence of expressions among the dictionary keys indicates multiple levels of index contractions. A nested dict displays nested contractions and may itself contain dicts from a deeper level. In practical calculations the summation in the deepest nested level must be calculated first so that the outer expression can access the resulting indexed object.

Examples

```
>>> from sympy.tensor.index_methods import get_contraction_structure
>>> from sympy import symbols, default_sort_key
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, k, l = map(Idx, ['i', 'j', 'k', 'l'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j])
{(i,): {x[i]*y[i]}, (j,): {A[j, j]}}
>>> get_contraction_structure(x[i]*y[j])
{None: {x[i]*y[j]}}
```

A multiplication of contracted factors results in nested dicts representing the internal contractions.

```
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d.keys(), key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```
>>> d[None]
{x[i, i]*y[j, j]}
```

Factors are contracted “first”:

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{(i,): {x[i, i]}}, {(j,): {y[j, j]}}]
```

A parenthesized Add object is also returned as a nested dictionary. The term containing the parenthesis is a Mul with a contraction among the arguments, so it will be found as a key in the result. It stores the dictionary resulting from a recursive call on the Add expression.

```
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d.keys(), key=default_sort_key)
[(A[i, j]*x[j] + y[i])*x[i], (i,)]
>>> d[(i,)]
{(A[i, j]*x[j] + y[i])*x[i]}
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[None: {y[i]}, (j,): {A[i, j]*x[j]}]
```

Powers with contractions in either base or exponent will also be found as keys in the dictionary, mapping to a list of results from recursive calls:

```
>>> d = get_contraction_structure(A[j, j]**A[i, i])
>>> d[None]
{A[j, j]**A[i, i]}
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): {A[j, j]}}
>>> nested_contractions[1]
{(i,): {A[i, i]}}
```

The description of the contraction structure may appear complicated when represented with a string in the above examples, but it is easy to iterate over:

```
>>> from sympy import Expr
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...         # treat outermost contractions here
```

`sympy.tensor.index_methods.get_indices(expr)`
Determine the outer indices of expression `expr`

By outer we mean indices that are not summation indices. Returns a set and a dict. The set contains outer indices and the dict contains information about index symmetries.

Examples

```
>>> from sympy.tensor.index_methods import get_indices
>>> from sympy import symbols
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, a, z = symbols('i j a z', integer=True)
```

The indices of the total expression is determined, Repeated indices imply a summation, for instance the trace of a matrix A:

```
>>> get_indices(A[i, i])
(set(), {})
```

In the case of many terms, the terms are required to have identical outer indices. Else an `IndexConformanceException` is raised.

```
>>> get_indices(x[i] + A[i, j]*y[j])
({i}, {})
```

Exceptions

An `IndexConformanceException` means that the terms are not compatible, e.g.

```
>>> get_indices(x[i] + y[j])
(...)
IndexConformanceException: Indices are not consistent: x(i) + y(j)
```

Warning: The concept of outer indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```
>>> get_indices((x[i] + A[i, j]*y[j])*x[j])
({i, j}, {})
```

This is correct and may appear convenient, but you need to be careful with this as SymPy will happily `.expand()` the product, if requested. The resulting expression would mix the outer `j` with the dummies inside the parenthesis, which makes it a different expression. To be on the safe side, it is best to avoid such ambiguities by using unique indices for all contractions that should be held separate.

Tensor

class `sympy.tensor.tensor._TensorManager`

Class to manage tensor properties.

Notes

Tensors belong to tensor commutation groups; each group has a label `comm`; there are predefined labels:

0 tensors commuting with any other tensor

1 tensors anticommuting among themselves

2 tensors not commuting, apart with those with `comm=0`

Other groups can be defined using `set_comm`; tensors in those groups commute with those with `comm=0`; by default they do not commute with any other group.

clear()

Clear the `TensorManager`.

comm_i2symbol(i)

Returns the symbol corresponding to the commutation group number.

comm_symbols2i(i)

get the commutation group number corresponding to `i`

`i` can be a symbol or a number or a string

If `i` is not already defined its commutation group number is set.

get_comm(i, j)

Return the commutation parameter for commutation group numbers i, j
see `_TensorManager.set_comm`

set_comm(i, j, c)

set the commutation parameter c for commutation groups i, j

Parameters i, j : symbols representing commutation groups

c : group commutation number

Notes

i, j can be symbols, strings or numbers, apart from 0, 1 and 2 which are reserved respectively for commuting, anticommuting tensors and tensors not commuting with any other group apart with the commuting tensors. For the remaining cases, use this method to set the commutation rules; by default c=None.

The group commutation number c is assigned in correspondence to the group commutation symbols; it can be

0 commuting

1 anticommuting

None no commutation property

Examples

G and GH do not commute with themselves and commute with each other; A is commuting.

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead, TensorManager
>>> Lorentz = TensorIndexType('Lorentz')
>>> i0,i1,i2,i3,i4 = tensor_indices('i0:5', Lorentz)
>>> A = tensorhead('A', [Lorentz], [[1]])
>>> G = tensorhead('G', [Lorentz], [[1]], 'Gcomm')
>>> GH = tensorhead('GH', [Lorentz], [[1]], 'GHcomm')
>>> TensorManager.set_comm('Gcomm', 'GHcomm', 0)
>>> (GH(i1)*G(i0)).canon_bp()
G(i0)*GH(i1)
>>> (G(i1)*G(i0)).canon_bp()
G(i1)*G(i0)
>>> (G(i1)*A(i0)).canon_bp()
A(i0)*G(i1)
```

set_comms(*args)

set the commutation group numbers c for symbols i, j

Parameters args : sequence of (i, j, c)

class sympy.tensor.tensor.TensorIndexType

A TensorIndexType is characterized by its name and its metric.

Parameters name : name of the tensor type

metric : metric symmetry or metric object or None

dim : dimension, it can be a symbol or an integer or None

eps_dim : dimension of the epsilon tensor

dummy_fmt : name of the head of dummy indices

Notes

The metric parameter can be: `metric = False` symmetric metric (in Riemannian geometry)

`metric = True` antisymmetric metric (for spinor calculus)

`metric = None` there is no metric

`metric` can be an object having `name` and `antisym` attributes.

If there is a metric the metric is used to raise and lower indices.

In the case of antisymmetric metric, the following raising and lowering conventions will be adopted:

$\psi(a) = g(a, b)\psi(-b)$; $\chi(-a) = \chi(b)g(-b, -a)$

$g(-a, b) = \delta(-a, b)$; $g(b, -a) = -\delta(a, -b)$

where $\delta(-a, b) = \delta(b, -a)$ is the Kronecker delta (see `TensorIndex` for the conventions on indices).

If there is no metric it is not possible to raise or lower indices; e.g. the index of the defining representation of $SU(N)$ is 'covariant' and the conjugate representation is 'contravariant'; for $N > 2$ they are linearly independent.

`eps_dim` is by default equal to `dim`, if the latter is an integer; else it can be assigned (for use in naive dimensional regularization); if `eps_dim` is not an integer `epsilon` is `None`.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> Lorentz.metric
metric(Lorentz, Lorentz)
```

Examples with metric components data added, this means it is working on a fixed basis:

```
>>> Lorentz.data = [1, -1, -1, -1]
>>> Lorentz
TensorIndexType(Lorentz, 0)
>>> Lorentz.data
[[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, -1]]
```

Attributes

name	
metric_name	(it is 'metric' or metric.name)
metric_antisym	
metric	(the metric tensor)
delta	(Kronecker delta)
epsilon	(the Levi-Civita epsilon tensor)
dim	
eps_dim	
dummy_fmt	
data	(a property to add ndarray values, to work in a specified basis.)

class sympy.tensor.tensor.TensorIndex

Represents an abstract tensor index.

Parameters **name** : name of the index, or True if you want it to be automatically assigned

tensor_type : TensorIndexType of the index

is_up : flag for contravariant index

Notes

Tensor indices are contracted with the Einstein summation convention.

An index can be in contravariant or in covariant form; in the latter case it is represented prepending a - to the index name.

Dummy indices have a name with head given by tensor_type._dummy_fmt

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorIndex, TensorSymmetry, _
↳TensorType, get_symmetric_group_sgs
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i = TensorIndex('i', Lorentz); i
i
>>> sym1 = TensorSymmetry(*get_symmetric_group_sgs(1))
>>> S1 = TensorType([Lorentz], sym1)
>>> A, B = S1('A,B')
>>> A(i)*B(-i)
A(L_0)*B(-L_0)
```

If you want the index name to be automatically assigned, just put True in the name field, it will be generated using the reserved character _ in front of its name, in order to avoid conflicts with possible existing indices:

```
>>> i0 = TensorIndex(True, Lorentz)
>>> i0
_i0
>>> i1 = TensorIndex(True, Lorentz)
```

(continues on next page)

(continued from previous page)

```
>>> i1
_i1
>>> A(i0)*B(-i1)
A(_i0)*B(-_i1)
>>> A(i0)*B(-i0)
A(L_0)*B(-L_0)
```

Attributes

name	
tensor_type	
is_up	

`sympy.tensor.tensor.tensor_indices(s, typ)`

Returns list of tensor indices given their names and their types

Parameters `s` : string of comma separated names of indices

`typ` : `TensorIndexType` of the indices

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
```

class `sympy.tensor.tensor.TensorSymmetry`

Monoterm symmetry of a tensor

Parameters `bsgs` : tuple (base, sgs) BSGS of the symmetry of the tensor

See also:

[sympy.combinatorics.tensor_can.get_symmetric_group_sgs](#) (page 293)

Notes

A tensor can have an arbitrary monoterm symmetry provided by its BSGS. Multiterm symmetries, like the cyclic symmetry of the Riemann tensor, are not covered.

Examples

Define a symmetric tensor

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorSymmetry, TensorType, _
↳ get_symmetric_group_sgs
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = TensorSymmetry(get_symmetric_group_sgs(2))
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Attributes

base	(base of the BSGS)
generators	(generators of the BSGS)
rank	(rank of the tensor)

`sympy.tensor.tensor.tensorsymmetry(*args)`

Return a TensorSymmetry object.

One can represent a tensor with any monotermin slot symmetry group using a BSGS.

`args` can be a BSGS `args[0]` base `args[1]` sgs

Usually tensors are in (direct products of) representations of the symmetric group; `args` can be a list of lists representing the shapes of Young tableaux

Notes

For instance: `[[1]]` vector `[[1]*n]` symmetric tensor of rank `n` `[[n]]` antisymmetric tensor of rank `n` `[[2, 2]]` monotermin slot symmetry of the Riemann tensor `[[1], [1]]` vector*vector `[[2], [1], [1]]` (antisymmetric tensor)*vector*vector

Notice that with the shape `[2, 2]` we associate only the monotermin symmetries of the Riemann tensor; this is an abuse of notation, since the shape `[2, 2]` corresponds usually to the irreducible representation characterized by the monotermin symmetries and by the cyclic symmetry.

Examples

Symmetric tensor using a Young tableau

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorType, tensorsymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1, 1])
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Symmetric tensor using a BSGS (base, strong generator set)

```
>>> from sympy.tensor.tensor import get_symmetric_group_sgs
>>> sym2 = tensorsymmetry(*get_symmetric_group_sgs(2))
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

class `sympy.tensor.tensor.TensorType`

Class of tensor types.

Parameters `index_types` : list of `TensorIndexType` of the tensor indices

`symmetry` : `TensorSymmetry` of the tensor

Examples

Define a symmetric tensor

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorsymmetry, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1, 1])
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Attributes

index_types	
symmetry	
types	(list of TensorIndexType without repetitions)

class sympy.tensor.tensor.TensorHead

Tensor head of the tensor

Parameters **name** : name of the tensor

typ : list of TensorIndexType

comm : commutation group number

Notes

A TensorHead belongs to a commutation group, defined by a symbol on number comm (see `_TensorManager.set_comm`); tensors in a commutation group have the same commutation properties; by default comm is 0, the group of the commuting tensors.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorhead, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> A = tensorhead('A', [Lorentz, Lorentz], [[1],[1]])
```

Examples with ndarray values, the components data assigned to the TensorHead object are assumed to be in a fully-contravariant representation. In case it is necessary to assign components data which represents the values of a non-fully covariant tensor, see the other examples.

```
>>> from sympy.tensor.tensor import tensor_indices, tensorhead
>>> Lorentz.data = [1, -1, -1, -1]
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
>>> A.data = [[j+2*i for j in range(4)] for i in range(4)]
```

in order to retrieve data, it is also necessary to specify abstract indices enclosed by round brackets, then numerical indices inside square brackets.

```
>>> A(i0, i1)[0, 0]
0
>>> A(i0, i1)[2, 3] == 3+2*2
True
```

Notice that square brackets create a valued tensor expression instance:

```
>>> A(i0, i1)
A(i0, i1)
```

To view the data, just type:

```
>>> A.data
[[0, 1, 2, 3], [2, 3, 4, 5], [4, 5, 6, 7], [6, 7, 8, 9]]
```

Turning to a tensor expression, covariant indices get the corresponding components data corrected by the metric:

```
>>> A(i0, -i1).data
[[0, -1, -2, -3], [2, -3, -4, -5], [4, -5, -6, -7], [6, -7, -8, -9]]
```

```
>>> A(-i0, -i1).data
[[0, -1, -2, -3], [-2, 3, 4, 5], [-4, 5, 6, 7], [-6, 7, 8, 9]]
```

while if all indices are contravariant, the ndarray remains the same

```
>>> A(i0, i1).data
[[0, 1, 2, 3], [2, 3, 4, 5], [4, 5, 6, 7], [6, 7, 8, 9]]
```

When all indices are contracted and components data are added to the tensor, accessing the data will return a scalar, no array object. In fact, arrays are dropped to scalars if they contain only one element.

```
>>> A(i0, -i0)
A(L_0, -L_0)
>>> A(i0, -i0).data
-18
```

It is also possible to assign components data to an indexed tensor, i.e. a tensor with specified covariant and contravariant components. In this example, the covariant components data of the Electromagnetic tensor are injected into A :

```
>>> from sympy import symbols
>>> Ex, Ey, Ez, Bx, By, Bz = symbols('E_x E_y E_z B_x B_y B_z')
>>> c = symbols('c', positive=True)
```

Let's define F , an antisymmetric tensor, we have to assign an antisymmetric matrix to it, because $[[2]]$ stands for the Young tableau representation of an antisymmetric set of two elements:

```
>>> F = tensorhead('A', [Lorentz, Lorentz], [[2]])
>>> F(-i0, -i1).data = [
... [0, Ex/c, Ey/c, Ez/c],
... [-Ex/c, 0, -Bz, By],
... [-Ey/c, Bz, 0, -Bx],
... [-Ez/c, -By, Bx, 0]]
```

Now it is possible to retrieve the contravariant form of the Electromagnetic tensor:

```
>>> F(i0, i1).data
[[0, -E_x/c, -E_y/c, -E_z/c], [E_x/c, 0, -B_z, B_y], [E_y/c, B_z, 0, -B_x], [E_z/
↪c, -B_y, B_x, 0]]
```

and the mixed contravariant-covariant form:

```
>>> F(i0, -i1).data
[[0, E_x/c, E_y/c, E_z/c], [E_x/c, 0, B_z, -B_y], [E_y/c, -B_z, 0, B_x], [E_z/c,
↪ -B_y, -B_x, 0]]
```

To convert the darray to a SymPy matrix, just cast:

```
>>> F.data.tomatrix()
Matrix([
[ 0, -E_x/c, -E_y/c, -E_z/c],
[E_x/c, 0, -B_z, B_y],
[E_y/c, B_z, 0, -B_x],
[E_z/c, -B_y, B_x, 0]])
```

Still notice, in this last example, that accessing components data from a tensor without specifying the indices is equivalent to assume that all indices are contravariant.

It is also possible to store symbolic components data inside a tensor, for example, define a four-momentum-like tensor:

```
>>> from sympy import symbols
>>> P = tensorhead('P', [Lorentz], [[1]])
>>> E, px, py, pz = symbols('E p_x p_y p_z', positive=True)
>>> P.data = [E, px, py, pz]
```

The contravariant and covariant components are, respectively:

```
>>> P(i0).data
[E, p_x, p_y, p_z]
>>> P(-i0).data
[E, -p_x, -p_y, -p_z]
```

The contraction of a 1-index tensor by itself is usually indicated by a power by two:

```
>>> P(i0)**2
E**2 - p_x**2 - p_y**2 - p_z**2
```

As the power by two is clearly identical to $P_\mu P^\mu$, it is possible to simply contract the TensorHead object, without specifying the indices

```
>>> P**2
E**2 - p_x**2 - p_y**2 - p_z**2
```

Attributes

name	
index_types	
rank	
types	(equal to typ.types)
symmetry	(equal to typ.symmetry)
comm	(commutation group)

commutes_with(other)

Returns 0 if self and other commute, 1 if they anticommute.

Returns None if self and other neither commute nor anticommute.

class `sympy.tensor.tensor.TensExpr`
Abstract base class for tensor expressions

Notes

A tensor expression is an expression formed by tensors; currently the sums of tensors are distributed.

A `TensExpr` can be a `TensAdd` or a `TensMul`.

`TensAdd` objects are put in canonical form using the Butler-Portugal algorithm for canonicalization under monotermin symmetries.

`TensMul` objects are formed by products of component tensors, and include a coefficient, which is a SymPy expression.

In the internal representation contracted indices are represented by (`ipos1`, `ipos2`, `icomp1`, `icomp2`), where `icomp1` is the position of the component tensor with contravariant index, `ipos1` is the slot which the index occupies in that component tensor.

Contracted indices are therefore nameless in the internal representation.

fun_eval(*index_tuples)
Return a tensor with free indices substituted according to `index_tuples`
`index_types` list of tuples (`old_index`, `new_index`)

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j); t
A(i, L_0)*B(-L_0, -j)
>>> t.fun_eval((i, k), (-j, l))
A(k, L_0)*B(-L_0, l)
```

get_matrix()
Returns ndarray components data as a matrix, if components data are available and ndarray dimension does not exceed 2.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorsymmetry, \
↳ TensorType
>>> from sympy import ones
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1]*2)
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> A = S2('A')
```

The tensor A is symmetric in its indices, as can be deduced by the [1, 1] Young tableau when constructing *sym2*. One has to be careful to assign symmetric component data to A, as the symmetry properties of data are currently not checked to be compatible with the defined tensor symmetry.

```
>>> from sympy.tensor.tensor import tensor_indices, tensorhead
>>> Lorentz.data = [1, -1, -1, -1]
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
>>> A.data = [[j+i for j in range(4)] for i in range(4)]
>>> A(i0, i1).get_matrix()
Matrix([
[0, 1, 2, 3],
[1, 2, 3, 4],
[2, 3, 4, 5],
[3, 4, 5, 6]])
```

It is possible to perform usual operation on matrices, such as the matrix multiplication:

```
>>> A(i0, i1).get_matrix()*ones(4, 1)
Matrix([
[ 6],
[10],
[14],
[18]])
```

class sympy.tensor.tensor.TensAdd

Sum of tensors

Parameters *free_args* : list of the free indices

Notes

Sum of more than one tensor are put automatically in canonical form.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorhead, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b = tensor_indices('a,b', Lorentz)
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(a) + q(a); t
p(a) + q(a)
>>> t(b)
p(b) + q(b)
```

Examples with components data added to the tensor expression:

```
>>> Lorentz.data = [1, -1, -1, -1]
>>> a, b = tensor_indices('a, b', Lorentz)
>>> p.data = [2, 3, -2, 7]
>>> q.data = [2, 3, -2, 7]
>>> t = p(a) + q(a); t
p(a) + q(a)
```

(continues on next page)

(continued from previous page)

```
>>> t(b)
p(b) + q(b)
```

The following are: $2^2 - 3^2 - 2^2 - 7^2 \Rightarrow -58$

```
>>> (p(a)*p(-a)).data
-58
>>> p(a)**2
-58
```

Attributes

<code>args</code>	(tuple of addends)
<code>rank</code>	(rank of the tensor)
<code>free_args</code>	(list of the free indices in sorted order)

`canon_bp()`

canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

`contract_metric(g)`

Raise or lower indices with the metric `g`

Parameters `g` : metric

contract_all : if True, eliminate all `g` which are contracted

Notes

see the `TensorIndexType` docstring for the contraction conventions

`fun_eval(*index_tuples)`

Return a tensor with free indices substituted according to `index_tuples`

Parameters `index_types` : list of tuples (old_index, new_index)

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j) + A(i, -j)
>>> t.fun_eval((i, k), (-j, l))
A(k, L_0)*B(l, -L_0) + A(k, l)
```

`substitute_indices(*index_tuples)`

Return a tensor with free indices substituted according to `index_tuples`

Parameters `index_types` : list of tuples (old_index, new_index)

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j); t
A(i, L_0)*B(-L_0, -j)
>>> t.substitute_indices((i,j), (j, k))
A(j, L_0)*B(-L_0, -k)
```

class sympy.tensor.tensor.TensMul

Product of tensors

Parameters **coeff** : SymPy coefficient of the tensor

args

Notes

args[0] list of TensorHead of the component tensors.

args[1] list of (ind, ipos, icomp) where ind is a free index, ipos is the slot position of ind in the icomp-th component tensor.

args[2] list of tuples representing dummy indices. (ipos1, ipos2, icomp1, icomp2) indicates that the contravariant dummy index is the ipos1-th slot position in the icomp1-th component tensor; the corresponding covariant index is in the ipos2 slot position in the icomp2-th component tensor.

Attributes

components	(list of TensorHead of the component tensors)
types	(list of nonrepeated TensorIndexType)
free	(list of (ind, ipos, icomp), see Notes)
dum	(list of (ipos1, ipos2, icomp1, icomp2), see Notes)
ext_rank	(rank of the tensor counting the dummy indices)
rank	(rank of the tensor)
coeff	(SymPy coefficient of the tensor)
free_args	(list of the free indices in sorted order)
is_canon_bp	(True if the tensor in in canonical form)

canon_bp()

Canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

Examples

```

>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> A = tensorhead('A', [Lorentz]*2, [[2]])
>>> t = A(m0, -m1)*A(m1, -m0)
>>> t.canon_bp()
-A(L_0, L_1)*A(-L_0, -L_1)
>>> t = A(m0, -m1)*A(m1, -m2)*A(m2, -m0)
>>> t.canon_bp()
0

```

contract_metric(g)

Raise or lower indices with the metric g

Parameters g : metric

Notes

see the TensorIndexType docstring for the contraction conventions

Examples

```

>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m0)*q(m1)*g(-m0, -m1)
>>> t.canon_bp()
metric(L_0, L_1)*p(-L_0)*q(-L_1)
>>> t.contract_metric(g).canon_bp()
p(L_0)*q(-L_0)

```

get_free_indices()

Returns the list of free indices of the tensor

The indices are listed in the order in which they appear in the component tensors.

Examples

```

>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m1)*g(m0,m2)
>>> t.get_free_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)

```

(continues on next page)

(continued from previous page)

```
>>> t2.get_free_indices()
[m2]
```

get_indices()

Returns the list of indices of the tensor

The indices are listed in the order in which they appear in the component tensors. The dummy indices are given a name which does not collide with the names of the free indices.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m1)*g(m0,m2)
>>> t.get_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_indices()
[L_0, -L_0, m2]
```

perm2tensor(g, is_canon_bp=False)

Returns the tensor corresponding to the permutation g

For further details, see the method in TIDS with the same name.

sorted_components()

Returns a tensor product with sorted components.

split()

Returns a list of tensors, whose product is self

Dummy indices contracted among different tensor components become free indices with the same name as the one used to represent the dummy indices.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(a,b)*B(-b,c)
>>> t
A(a, L_0)*B(-L_0, c)
>>> t.split()
[A(a, L_0), B(-L_0, c)]
```

sympy.tensor.tensor.canon_bp(p)

Butler-Portugal canonicalization

`sympy.tensor.tensor.tensor_mul(*a)`

product of tensors

`sympy.tensor.tensor.riemann_cyclic_replace(t_r)`

replace Riemann tensor with an equivalent expression

$R(m,n,p,q) \rightarrow \frac{2}{3}R(m,n,p,q) - \frac{1}{3}R(m,q,n,p) + \frac{1}{3}R(m,p,n,q)$

`sympy.tensor.tensor.riemann_cyclic(t2)`

replace each Riemann tensor with an equivalent expression satisfying the cyclic identity.

This trick is discussed in the reference guide to Cadabra.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead, u
↪riemann_cyclic
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> R = tensorhead('R', [Lorentz]*4, [[2, 2]])
>>> t = R(i,j,k,l)*(R(-i,-j,-k,-l) - 2*R(-i,-k,-j,-l))
>>> riemann_cyclic(t)
0
```

5.36 Utilities

This module contains some general purpose utilities that are used across SymPy.

Contents:

5.36.1 Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the [Tensor Module](#) (page 1360). Here is a simple example that shows how to setup a binary routine that calculates a matrix-vector product.

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy import symbols, IndexedBase, Idx, Eq
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
>>> m, n = symbols('m n', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j]); instruction
Eq(y[i], A[i, j]*x[j])
```

Because the code printers treat Indexed objects with repeated indices as a summation, the above equality instance will be translated to low-level code for a matrix vector product. This is how you tell SymPy to generate the code, compile it and wrap it as a python function:

```
>>> matvec = autowrap(instruction)
```

That's it. Now let's test it with some numpy arrays. The default wrapper backend is f2py. The wrapper function it provides is set up to accept python lists, which it will silently convert to numpy arrays. So we can test the matrix vector product like this:

```
>>> M = [[0, 1],
...       [1, 0]]
>>> matvec(M, [2, 3])
[ 3.  2.]
```

Implementation details

The autowrap module is implemented with a backend consisting of CodeWrapper objects. The base class CodeWrapper takes care of details about module name, filenames and options. It also contains the driver routine, which runs through all steps in the correct order, and also takes care of setting up and removing the temporary working directory.

The actual compilation and wrapping is done by external resources, such as the system installed f2py command. The Cython backend runs a distutils setup script in a subprocess. Subclasses of CodeWrapper takes care of these backend-dependent details.

API Reference

Module for compiling codegen output, and wrap the binary for use in python.

Note: To use the autowrap module it must first be imported

```
>>> from sympy.utilities.autowrap import autowrap
```

This module provides a common interface for different external backends, such as f2py, fwrap, Cython, SWIG(?) etc. (Currently only f2py and Cython are implemented) The goal is to provide access to compiled binaries of acceptable performance with a one-button user interface, i.e.

```
>>> from sympy.abc import x,y
>>> expr = ((x - y)**(25)).expand()
>>> binary_callable = autowrap(expr)
>>> binary_callable(1, 2)
-1.0
```

The callable returned from autowrap() is a binary python function, not a SymPy object. If it is desired to use the compiled function in symbolic expressions, it is better to use binary_function() which returns a SymPy Function object. The binary callable is attached as the `_imp_` attribute and invoked when a numerical evaluation is requested with `evalf()`, or with `lambdify()`.

```
>>> from sympy.utilities.autowrap import binary_function
>>> f = binary_function('f', expr)
>>> 2*f(x, y) + y
y + 2*f(x, y)
>>> (2*f(x, y) + y).evalf(2, subs={x: 1, y:2})
0.e-110
```

The idea is that a SymPy user will primarily be interested in working with mathematical expressions, and should not have to learn details about wrapping tools in order to evaluate expressions numerically, even if they are computationally expensive.

When is this useful?

1. For computations on large arrays, Python iterations may be too slow, and depending on the mathematical expression, it may be difficult to exploit the advanced index operations provided by NumPy.
2. For really long expressions that will be called repeatedly, the compiled binary should be significantly faster than SymPy's `.evalf()`
3. If you are generating code with the codegen utility in order to use it in another project, the automatic python wrappers let you test the binaries immediately from within SymPy.
4. To create customized ufuncs for use with numpy arrays. See `ufuncify`.

When is this module NOT the best approach?

1. If you are really concerned about speed or memory optimizations, you will probably get better results by working directly with the wrapper tools and the low level code. However, the files generated by this utility may provide a useful starting point and reference code. Temporary files will be left intact if you supply the keyword `tempdir="path/to/files/"`.
2. If the array computation can be handled easily by numpy, and you don't need the binaries for another project.

class `sympy.utilities.autowrap.CodeWrapper`(generator, filepath=None, flags=[],
verbose=False)

Base Class for code wrappers

class `sympy.utilities.autowrap.CythonCodeWrapper`(*args, **kwargs)
Wrapper that uses Cython

dump_pyx(routines, f, prefix)

Write a Cython file with python wrappers

This file contains all the definitions of the routines in c code and refers to the header file.

Arguments

routines List of Routine instances

f File-like object to write the file to

prefix The filename prefix, used to refer to the proper header file. Only the base-name of the prefix is used.

class `sympy.utilities.autowrap.DummyWrapper`(generator, filepath=None, flags=[],
verbose=False)

Class used for testing independent of backends

class `sympy.utilities.autowrap.F2PyCodeWrapper`(*args, **kwargs)
Wrapper that uses f2py

class `sympy.utilities.autowrap.UfuncifyCodeWrapper`(*args, **kwargs)
Wrapper for Ufuncify

dump_c(routines, f, prefix, funcname=None)

Write a C file with python wrappers

This file contains all the definitions of the routines in c code.

Arguments

routines List of Routine instances

f File-like object to write the file to

prefix The filename prefix, used to name the imported module.

funcname Name of the main function to be returned.

```
sympy.utilities.autowrap.autowrap(expr, language=None, backend='f2py', tempdir=None, args=None, flags=None, verbose=False, helpers=None, code_gen=None, **kwargs)
```

Generates python callable binaries based on the math expression.

Parameters **expr**

The SymPy expression that should be wrapped as a binary routine.

language : string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.

backend : string, optional

Backend used to wrap the generated code. Either 'f2py' [default], or 'cython'.

tempdir : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

args : iterable, optional

An ordered iterable of symbols. Specifies the argument sequence for the function.

flags : iterable, optional

Additional option flags that will be passed to the backend.

verbose : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

helpers : iterable, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in the helpers iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<function_name>, <sympy_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

code_gen : CodeGen instance

An instance of a CodeGen subclass. Overrides language.

include_dirs : [string]

A list of directories to search for C/C++ header files (in Unix form for portability).

library_dirs : [string]

A list of directories to search for C/C++ libraries at link time.

libraries : [string]

A list of library names (not filenames or paths) to link against.

extra_compile_args : [string]

Any extra platform- and compiler-specific information to use when compiling the source files in 'sources'. For platforms and compilers where "command line" makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.

extra_link_args : [string]

Any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for 'extra_compile_args'.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

`sympy.utilities.autowrap.binary_function(symfunc, expr, **kwargs)`

Returns a sympy function with `expr` as binary implementation

This is a convenience function that automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with `implemented_function()`.

Parameters **symfunc** : sympy Function

The function to bind the callable to.

expr : sympy Expression

The expression used to generate the function.

kwargs : dict

Any kwargs accepted by `autowrap`.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.utilities.autowrap import binary_function
>>> expr = ((x - y)**(25)).expand()
>>> f = binary_function('f', expr)
>>> type(f)
```

(continues on next page)

(continued from previous page)

```

<class 'sympy.core.function.UndefinedFunction'>
>>> 2*f(x, y)
2*f(x, y)
>>> f(x, y).evalf(2, subs={x: 1, y: 2})
-1.0

```

`sympy.utilities.autowrap.ufuncify`(args, expr, language=None, backend='numpy',
tempdir=None, flags=None, verbose=False,
helpers=None, **kwargs)

Generates a binary function that supports broadcasting on numpy arrays.

Parameters args : iterable

Either a Symbol or an iterable of symbols. Specifies the argument sequence for the function.

expr

A SymPy expression that defines the element wise operation.

language : string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.

backend : string, optional

Backend used to wrap the generated code. Either 'numpy' [default], 'cython', or 'f2py'.

tempdir : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

flags : iterable, optional

Additional option flags that will be passed to the backend.

verbose : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

helpers : iterable, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in the helpers iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<function_name>, <sympy_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

kwargs : dict

These kwargs will be passed to autowrap if the *f2py* or *cython* backend is used and ignored if the *numpy* backend is used.

References

[1] <http://docs.scipy.org/doc/numpy/reference/ufuncs.html>

Examples

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.abc import x, y
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f)
<class 'numpy.ufunc'>
>>> f([1, 2, 3], 2)
array([ 3.,  6., 11.])
>>> f(np.arange(5), 3)
array([ 3.,  4.,  7., 12., 19.])
```

For the 'f2py' and 'cython' backends, inputs are required to be equal length 1-dimensional arrays. The 'f2py' backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```
>>> f_fortran = ufuncify((x, y), y + x**2, backend='f2py')
>>> f_fortran(1, 2)
array([ 3.])
>>> f_fortran(np.array([1, 2, 3]), np.array([1.0, 2.0, 3.0]))
array([ 2.,  6., 12.])
>>> f_cython = ufuncify((x, y), y + x**2, backend='Cython')
>>> f_cython(1, 2)
Traceback (most recent call last):
...
TypeError: Argument '_x' has incorrect type (expected numpy.ndarray, got int)
>>> f_cython(np.array([1.0]), np.array([2.0]))
array([ 3.])
```

Note

The default backend ('numpy') will create actual instances of `numpy.ufunc`. These support ndimensional broadcasting, and implicit type conversion. Use of the other backends will result in a "ufunc-like" function, which requires equal length 1-dimensional arrays for all arguments, and will not perform any type conversions.

5.36.2 Codegen

This module provides functionality to generate directly compilable code from SymPy expressions. The `codegen` function is the user interface to the code generation functionality in SymPy. Some details of the implementation is given below for advanced users that may want to use the framework directly.

Note: The `codegen` callable is not in the `sympy` namespace automatically, to use it you must first execute

```
>>> from sympy.utilities.codegen import codegen
```

Implementation Details

Here we present the most important pieces of the internal structure, as advanced users may want to use it directly, for instance by subclassing a code generator for a specialized application. **It is very likely that you would prefer to use the `codegen()` function documented above.**

Basic assumptions:

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

Routine

The Routine class is a very important piece of the codegen module. Viewing the codegen utility as a translator of mathematical expressions into a set of statements in a programming language, the Routine instances are responsible for extracting and storing information about how the math can be encapsulated in a function call. Thus, it is the Routine constructor that decides what arguments the routine will need and if there should be a return value.

API Reference

module for generating C, C++, Fortran77, Fortran90, Julia, Rust and Octave/Matlab routines that evaluate sympy expressions. This module is work in progress. Only the milestones with a '+' character in the list below have been completed.

— How is `sympy.utilities.codegen` different from `sympy.printing.ccode`? —

We considered the idea to extend the printing routines for sympy functions in such a way that it prints complete compilable code, but this leads to a few unsurmountable issues that can only be tackled with dedicated code generator:

- For C, one needs both a code and a header file, while the printing routines generate just one string. This code generator can be extended to support .pyf files for f2py.
- SymPy functions are not concerned with programming-technical issues, such as input, output and input-output arguments. Other examples are contiguous or non-contiguous arrays, including headers of other libraries such as gsl or others.
- It is highly interesting to evaluate several sympy functions in one C routine, eventually sharing common intermediate results with the help of the cse routine. This is more than just printing.
- From the programming perspective, expressions with constants should be evaluated in the code generator as much as possible. This is different for printing.

— Basic assumptions —

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.

- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

— Milestones —

- First working version with scalar input arguments, generating C code, tests
- Friendly functions that are easier to use than the rigorous Routine/CodeGen workflow.
- Integer and Real numbers as input and output
- Output arguments
- InputOutput arguments
- Sort input/output arguments properly
- Contiguous array arguments (numpy matrices)
- Also generate .pyf code for f2py (in autowrap module)
- Isolate constants and evaluate them beforehand in double precision
- Fortran 90
- Octave/Matlab
- Common Subexpression Elimination
- User defined comments in the generated code
- Optional extra include lines for libraries/objects that can eval special functions
- Test other C compilers and libraries: gcc, tcc, libtcc, gcc+gsl, ...
- Contiguous array arguments (sympy matrices)
- Non-contiguous array arguments (sympy matrices)
- ccode must raise an error when it encounters something that can not be translated into c. `ccode(integrate(sin(x)/x, x))` does not make sense.
- Complex numbers as input and output
- A default complex datatype
- Include extra information in the header: date, user, hostname, sha1 hash, ...
- Fortran 77
- C++
- Python
- Julia
- Rust
- ...

class `sympy.utilities.codegen.Routine`(name, arguments, results, local_vars, global_vars)

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or

Fortran. Another example: Fortran and Python support complex numbers, while C does not.

result_variables

Returns a list of OutputArgument, InOutArgument and Result.

If return values are present, they are at the end of the list.

variables

Returns a set of all variables possibly used in the routine.

For routines with unnamed return values, the dummies that may or may not be used will be included in the set.

class sympy.utilities.codegen.**DataType**(cname, fname, pname, jlname, octname, rsname)

Holds strings for a certain datatype in different languages.

sympy.utilities.codegen.**get_default_datatype**(expr)

Derives an appropriate datatype based on the expression.

class sympy.utilities.codegen.**Argument**(name, datatype=None, dimensions=None, precision=None)

An abstract Argument data structure: a name and a data type.

This structure is refined in the descendants below.

class sympy.utilities.codegen.**OutputArgument**(name, result_var, expr, datatype=None, dimensions=None, precision=None)

OutputArgument are always initialized in the routine.

class sympy.utilities.codegen.**Result**(expr, name=None, result_var=None, datatype=None, dimensions=None, precision=None)

An expression for a return value.

The name result is used to avoid conflicts with the reserved word “return” in the python language. It is also shorter than ReturnValue.

These may or may not need a name in the destination (e.g., “return(x*y)” might return a value without ever naming it).

class sympy.utilities.codegen.**CodeGen**(project='project', cse=False)

Abstract class for the code generators.

Attributes

printer	

dump_code(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

routine(name, expr, argument_sequence=None, global_vars=None)

Creates an Routine object that is appropriate for this language.

This implementation is appropriate for at least C/Fortran. Subclasses can override this if necessary.

Here, we assume at most one return value (the l-value) which must be scalar. Additional outputs are OutputArguments (e.g., pointers on right-hand-side or pass-by-reference). Matrices are always returned via OutputArguments. If argument_sequence is None, arguments will be ordered alphabetically, but with all InputArguments first, and then OutputArgument and InOutArguments.

write(routines, prefix, to_files=False, header=True, empty=True)

Writes all the source code files for the given routines.

The generated source is returned as a list of (filename, contents) tuples, or is written to files (see below). Each filename consists of the given prefix, appended with an appropriate extension.

Parameters routines : list

A list of Routine instances to be written

prefix : string

The prefix for the output files

to_files : bool, optional

When True, the output is written to files. Otherwise, a list of (filename, contents) tuples is returned. [default: False]

header : bool, optional

When True, a header comment is included on top of each source file.
[default: True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default: True]

class sympy.utilities.codegen.CCodeGen(project='project', printer=None, preprocessor_statements=None, cse=False)

Generator for C code.

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.c and <prefix>.h respectively.

Attributes

printer	

dump_c(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

dump_h(routines, f, prefix, header=True, empty=True)

Writes the C header file.

This file contains all the function declarations.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to construct the include guards. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

get_prototype(routine)

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an CodeGenError is raised.

See: http://en.wikipedia.org/wiki/Function_prototype

class sympy.utilities.codegen.FCodeGen(project='project', printer=None)

Generator for Fortran 95 code

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.f90 and <prefix>.h respectively.

Attributes

printer	

dump_f95(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

dump_h(routines, f, prefix, header=True, empty=True)

Writes the interface to a header file.

This file contains all the function declarations.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

get_interface(routine)

Returns a string for the function interface.

The routine should have a single result object, which can be None. If the routine has multiple result objects, a `CodeGenError` is raised.

See: http://en.wikipedia.org/wiki/Function_prototype

class `sympy.utilities.codegen.JuliaCodeGen`(project='project', printer=None)
Generator for Julia code.

The `.write()` method inherited from `CodeGen` will output a code file `<prefix>.jl`.

Attributes

printer	

dump_jl(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters **routines** : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

routine(name, expr, argument_sequence, global_vars)

Specialized Routine creation for Julia.

class `sympy.utilities.codegen.OctaveCodeGen`(project='project', printer=None)
Generator for Octave code.

The `.write()` method inherited from `CodeGen` will output a code file `<prefix>.m`.

Octave `.m` files usually contain one function. That function name should match the file-name (prefix). If you pass multiple `name_expr` pairs, the latter ones are presumed to be private functions accessed by the primary function.

You should only pass inputs to `argument_sequence`: outputs are ordered according to their order in `name_expr`.

Attributes

printer	

dump_m(routines, f, prefix, header=True, empty=True, inline=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

routine(name, expr, argument_sequence, global_vars)

Specialized Routine creation for Octave.

class sympy.utilities.codegen.**RustCodeGen**(project='project', printer=None)

Generator for Rust code.

The .write() method inherited from CodeGen will output a code file <prefix>.rs

Attributes

printer	

dump_rs(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file.
[default : True]

empty : bool, optional

When True, empty lines are included to structure the source files.
[default : True]

get_prototype(routine)

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an `CodeGenError` is raised.

See: http://en.wikipedia.org/wiki/Function_prototype

routine(name, expr, argument_sequence, global_vars)

Specialized Routine creation for Rust.

```
sympy.utilities.codegen.codegen(name_expr, language=None, prefix=None,
                                project='project', to_files=False, header=True,
                                empty=True, argument_sequence=None,
                                global_vars=None, standard=None,
                                code_gen=None, printer=None)
```

Generate source code for expressions in a given language.

Parameters **name_expr** : tuple, or list of tuples

A single (name, expression) tuple or a list of (name, expression) tuples. Each tuple corresponds to a routine. If the expression is an equality (an instance of class `Equality`) the left hand side is considered an output argument. If expression is an iterable, then the routine will have multiple outputs.

language : string,

A string that indicates the source code language. This is case insensitive. Currently, 'C', 'F95' and 'Octave' are supported. 'Octave' generates code compatible with both Octave and Matlab.

prefix : string, optional

A prefix for the names of the files that contain the source code. Language-dependent suffixes will be appended. If omitted, the name of the first `name_expr` tuple is used.

project : string, optional

A project name, used for making unique preprocessor instructions.
[default: "project"]

to_files : bool, optional

When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]

header : bool, optional

When True, a header is written on top of each source file. [default: True]

empty : bool, optional

When True, empty lines are used to structure the code. [default: True]

argument_sequence : iterable, optional

Sequence of arguments for the routine in a preferred order. A CodeGenError is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.

global_vars : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

standard : string

code_gen : CodeGen instance

An instance of a CodeGen subclass. Overrides language.

Examples

```
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     ("f", x+y*z), "C89", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
    double f_result;
    f_result = x + y*z;
    return f_result;
}

>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT_TEST_H
#define PROJECT_TEST_H
double f(double x, double y, double z);
#endif
```

Another example using Equality objects to give named outputs. Here the filename (prefix) is taken from the first (name, expr) pair.

```
>>> from sympy.abc import f, g
>>> from sympy import Eq
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     [("myfcn", x + y), ("fcn2", [Eq(f, 2*x), Eq(g, y)])],
...     "C99", header=False, empty=False)
>>> print(c_name)
myfcn.c
>>> print(c_code)
#include "myfcn.h"
#include <math.h>
double myfcn(double x, double y) {
```

(continues on next page)

(continued from previous page)

```

double myfcn_result;
myfcn_result = x + y;
return myfcn_result;
}
void fcn2(double x, double y, double *f, double *g) {
    (*f) = 2*x;
    (*g) = y;
}

```

If the generated function(s) will be part of a larger project where various global variables have been defined, the 'global_vars' option can be used to remove the specified variables from the function signature

```

>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(f_name, f_code), header] = codegen(
...     ("f", x+y*z), "F95", header=False, empty=False,
...     argument_sequence=(x, y), global_vars=(z,))
>>> print(f_code)
REAL*8 function f(x, y)
implicit none
REAL*8, intent(in) :: x
REAL*8, intent(in) :: y
f = x + y*z
end function

```

`sympy.utilities.codegen.make_routine(name, expr, argument_sequence=None, global_vars=None, language='F95')`

A factory that makes an appropriate Routine from an expression.

Parameters **name** : string

The name of this routine in the generated code.

expr : expression or list/tuple of expressions

A SymPy expression that the Routine instance will represent. If given a list or tuple of expressions, the routine will be considered to have multiple return values and/or output arguments.

argument_sequence : list or tuple, optional

List arguments for the routine in a preferred order. If omitted, the results are language dependent, for example, alphabetical order or in the same order as the given expressions.

global_vars : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

language : string, optional

Specify a target language. The Routine itself should be language-agnostic but the precise way one is created, error checking, etc depend on the language. [default: "F95"].

A decision about whether to use output arguments or return values is made

depending on both the language and the particular mathematical expressions.

For an expression of type Equality, the left hand side is typically made into an OutputArgument (or perhaps an InOutArgument if appropriate).

Otherwise, typically, the calculated expression is made a return values of the routine.

Examples

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.abc import x, y, f, g
>>> from sympy import Eq
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars
set()
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```
>>> from sympy import Matrix
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x, 2]])])
>>> [arg.result_var for arg in r.results]
[result_5397460570204848505]
>>> [arg.expr for arg in r.results]
[x*y]
>>> [arg.name for arg in r.arguments]
[x, y, f, g, out_8598435338387848786]
```

We can examine the various arguments more closely:

```
>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
...                                     InOutArgument)
>>> [a.name for a in r.arguments if isinstance(a, InputArgument)]
[x, y]
```

```
>>> [a.name for a in r.arguments if isinstance(a, OutputArgument)]
[f, out_8598435338387848786]
>>> [a.expr for a in r.arguments if isinstance(a, OutputArgument)]
[1, Matrix([[x, 2]])]
```

```
>>> [a.name for a in r.arguments if isinstance(a, InOutArgument)]
[g]
>>> [a.expr for a in r.arguments if isinstance(a, InOutArgument)]
[g + x]
```

5.36.3 Decorator

Useful utility decorators.

`sympy.utilities.decorator.conserve_mpmath_dps(func)`

After the function finishes, resets the value of `mpmath.mp.dps` to the value it had before the function was run.

`sympy.utilities.decorator.doctest_depends_on(exe=None, modules=None, disable_viewers=None)`

Adds metadata about the dependencies which need to be met for doctesting the docstrings of the decorated objects.

`sympy.utilities.decorator.memoize_property(storage)`

Create a property, where the lookup is stored in `storage`

`class sympy.utilities.decorator.no_attrs_in_subclass(cls, f)`

Don't 'inherit' certain attributes from a base class

```
>>> from sympy.utilities.decorator import no_attrs_in_subclass
```

```
>>> class A(object):
...     x = 'test'
```

```
>>> A.x = no_attrs_in_subclass(A, A.x)
```

```
>>> class B(A):
...     pass
```

```
>>> hasattr(A, 'x')
True
>>> hasattr(B, 'x')
False
```

`sympy.utilities.decorator.public(obj)`

Append `obj`'s name to global `__all__` variable (call site).

By using this decorator on functions or classes you achieve the same goal as by filling `__all__` variables manually, you just don't have to repeat yourself (object's name). You also know if object is public at definition site, not at some random location (where `__all__` was set).

Note that in multiple decorator setup (in almost all cases) `@public` decorator must be applied before any other decorators, because it relies on the pointer to object's global namespace. If you apply other decorators first, `@public` may end up modifying the wrong namespace.

Examples

```
>>> from sympy.utilities.decorator import public
```

```
>>> __all__
Traceback (most recent call last):
...
NameError: name '__all__' is not defined
```

```
>>> @public
... def some_function():
...     pass
```

```
>>> __all__
['some_function']
```

`sympy.utilities.decorator.threaded(func)`

Apply `func` to sub-elements of an object, including `Add`.

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 1410) decorator allows threading over elements of `Add` class. If this behavior is not desirable use `xthreaded()` (page 1410) decorator.

Functions using this decorator must have the following signature:

```
@threaded
def function(expr, *args, **kwargs):
```

`sympy.utilities.decorator.threaded_factory(func, use_add)`

A factory for threaded decorators.

`sympy.utilities.decorator.xthreaded(func)`

Apply `func` to sub-elements of an object, excluding `Add`.

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 1410) decorator disallows threading over elements of `Add` class. If this behavior is not desirable use `threaded()` (page 1410) decorator.

Functions using this decorator must have the following signature:

```
@xthreaded
def function(expr, *args, **kwargs):
```

5.36.4 Enumerative

This module includes functions and classes for enumerating and counting multiset partitions.

`sympy.utilities.enumerative.multiset_partitions_taocp(multiplicities)`

Enumerates partitions of a multiset.

Parameters `multiplicities`

list of integer multiplicities of the components of the multiset.

Yields `state`

Internal data structure which encodes a particular partition. This output is then usually processed by a visitor function which combines the information from this data structure with the components themselves to produce an actual partition.

Unless they wish to create their own visitor function, users will have little need to look inside this data structure. But, for reference, it is a 3-element list with components:

f is a frame array, which is used to divide pstack into parts.

lpart points to the base of the topmost part.

pstack is an array of PartComponent objects.

The state output offers a peek into the internal data structures of the enumeration function. The client should treat this as read-only; any modification of the data structure will cause unpredictable (and almost certainly incorrect) results. Also, the components of state are modified in place at each iteration. Hence, the visitor must be called at each loop iteration. Accumulating the state instances and processing them later will not work.

See also:

sympy.utilities.iterables.multiset_partitions (page 1425) Takes a multiset as input and directly yields multiset partitions. It dispatches to a number of functions, including this one, for implementation. Most users will find it more convenient to use than multiset_partitions_taocp.

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> # variables components and multiplicities represent the multiset 'abb'
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(list_visitor(state, components) for state in states)
[[['a', 'b', 'b']],
 [['a', 'b'], ['b']],
 [['a'], ['b', 'b']],
 [['a'], ['b'], ['b']]]
```

sympy.utilities.enumerative.factoring_visitor(state, primes)

Use with multiset_partitions_taocp to enumerate the ways a number can be expressed as a product of factors. For this usage, the exponents of the prime factors of a number are arguments to the partition enumerator, while the corresponding prime factors are input here.

Examples

To enumerate the factorings of a number we can think of the elements of the partition as being the prime factors and the multiplicities as being their exponents.

```
>>> from sympy.utilities.enumerative import factoring_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> from sympy import factorint
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
```

(continues on next page)

(continued from previous page)

```
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(factoring_visitor(state, primes) for state in states)
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]
```

`sympy.utilities.enumerative.list_visitor(state, components)`

Return a list of lists to represent the partition.

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc') # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3]) # for multiset '1 2 2 3'
[[1, 2, 2], [3]]
```

The approach of the function `multiset_partitions_taocp` is extended and generalized by the class `MultisetPartitionTraverser`.

class `sympy.utilities.enumerative.MultisetPartitionTraverser`

Has methods to enumerate and count the partitions of a multiset.

This implements a refactored and extended version of Knuth's algorithm 7.1.2.5M [AOCp] (page 1872)."

The enumeration methods of this class are generators and return data structures which can be interpreted by the same visitor functions used for the output of `multiset_partitions_taocp`.

See also:

`multiset_partitions_taocp` (page 1410), `sympy.utilities.iterables.multiset_partitions`

References

[AOCp] (page 1872), [Factorisatio] (page 1872), [Yorgey] (page 1872)

Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4,4,4,2])
127750
>>> m.count_partitions([3,3,3])
686
```

count_partitions(multiplicities)

Returns the number of partitions of a multiset whose components have the multiplicities given in `multiplicities`.

For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the `MultisetPartitionTraverser` object and persists across calls. If the user does not expect to call `count_partitions` for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to `count_partitions`, so it may be advantageous not to clear the object.

Notes

If one looks at the workings of Knuth's algorithm M [AOCF] (page 1872), it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

1. This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)
2. Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no `_range` version of `count_partitions`. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9,8,2])
288716
>>> m.count_partitions([2,2])
```

(continues on next page)

(continued from previous page)

```
9
>>> del m
```

enum_all(multiplicities)

Enumerate the partitions of a multiset.

See also:

multiset_partitions_taocp (page 1410) which provides the same result as this method, but is about twice as fast. Hence, `enum_all` is primarily useful for testing. Also see the function for a discussion of states and visitors.

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_all([2,2])
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
 [['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'a'], ['b'], ['b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']],
 [['a', 'b'], ['a'], ['b']],
 [['a'], ['a'], ['b', 'b']],
 [['a'], ['a'], ['b'], ['b']]]
```

enum_large(multiplicities, lb)

Enumerate the partitions of a multiset with $lb < \text{num}(\text{parts})$

Equivalent to `enum_range(multiplicities, lb, sum(multiplicities))`

Parameters multiplicities

list of multiplicities of the components of the multiset.

lb

Number of parts in the partition must be greater than this lower bound.

See also:

`enum_all` (page 1414), `enum_small` (page 1415), `enum_range` (page 1415)

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_large([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a'], ['b'], ['b']],
 [['a', 'b'], ['a'], ['b']],
```

(continues on next page)

(continued from previous page)

```
[[ 'a' ], [ 'a' ], [ 'b', 'b' ]],
[[ 'a' ], [ 'a' ], [ 'b' ], [ 'b' ]]]
```

enum_range(multiplicities, lb, ub)

Enumerate the partitions of a multiset with $lb < \text{num}(\text{parts}) \leq ub$.

In particular, if partitions with exactly k parts are desired, call with `(multiplicities, k - 1, k)`. This method generalizes `enum_all`, `enum_small`, and `enum_large`.

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2,2], 1, 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[ 'a', 'a', 'b' ], [ 'b' ]],
[[ 'a', 'a' ], [ 'b', 'b' ]],
[[ 'a', 'b', 'b' ], [ 'a' ]],
[[ 'a', 'b' ], [ 'a', 'b' ]]
```

enum_small(multiplicities, ub)

Enumerate multiset partitions with no more than `ub` parts.

Equivalent to `enum_range(multiplicities, 0, ub)`

Parameters **multiplicities**

list of multiplicities of the components of the multiset.

ub

Maximum number of parts

See also:

[enum_all](#) (page 1414), [enum_large](#) (page 1414), [enum_range](#) (page 1415)

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[ 'a', 'a', 'b', 'b' ],
[[ 'a', 'a', 'b' ], [ 'b' ]],
[[ 'a', 'a' ], [ 'b', 'b' ]],
[[ 'a', 'b', 'b' ], [ 'a' ]],
[[ 'a', 'b' ], [ 'a', 'b' ]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth [\[AOCP\]](#) (page 1872).

5.36.5 Iterables

cartes

Returns the cartesian product of sequences as a generator.

Examples::

```
>>> from sympy.utilities.iterables import cartes
>>> list(cartes([1,2,3], 'ab'))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

variations

variations(seq, n) Returns all the variations of the list of size n.

Has an optional third argument. Must be a boolean value and makes the method return the variations with repetition if set to True, or the variations without repetition if set to False.

Examples::

```
>>> from sympy.utilities.iterables import variations
>>> list(variations([1,2,3], 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(variations([1,2,3], 2, True))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

partitions

Although the combinatorics module contains Partition and IntegerPartition classes for investigation and manipulation of partitions, there are a few functions to generate partitions that can be used as low-level tools for routines: partitions and multiset_partitions. The former gives integer partitions, and the latter gives enumerated partitions of elements. There is also a routine kbins that will give a variety of permutations of partitions.

partitions:

```
>>> from sympy.utilities.iterables import partitions
>>> [p.copy() for s, p in partitions(7, m=2, size=True) if s == 2]
[{1: 1, 6: 1}, {2: 1, 5: 1}, {3: 1, 4: 1}]
```

multiset_partitions:

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> [p for p in multiset_partitions(3, 2)]
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
>>> [p for p in multiset_partitions([1, 1, 1, 2], 2)]
[[[1, 1, 1], [2]], [[1, 1, 2], [1]], [[1, 1], [1, 2]]]
```

kbins:

```
>>> from sympy.utilities.iterables import kbins
>>> def show(k):
...     rv = []
...     for p in k:
```

(continues on next page)

(continued from previous page)

```

...     rv.append(','.join([''.join(j) for j in p]))
...     return sorted(rv)
...
>>> show(kbins("ABCD", 2))
['A,BCD', 'AB,CD', 'ABC,D']
>>> show(kbins("ABC", 2))
['A,BC', 'AB,C']
>>> show(kbins("ABC", 2, ordered=0)) # same as multiset_partitions
['A,BC', 'AB,C', 'AC,B']
>>> show(kbins("ABC", 2, ordered=1))
['A,BC', 'A,CB',
 'B,AC', 'B,CA',
 'C,AB', 'C,BA']
>>> show(kbins("ABC", 2, ordered=10))
['A,BC', 'AB,C', 'AC,B',
 'B,AC', 'BC,A',
 'C,AB']
>>> show(kbins("ABC", 2, ordered=11))
['A,BC', 'A,CB', 'AB,C', 'AC,B',
 'B,AC', 'B,CA', 'BA,C', 'BC,A',
 'C,AB', 'C,BA', 'CA,B', 'CB,A']

```

Docstring

`sympy.utilities.iterables.binary_partitions(n)`

Generates the binary partition of n .

A binary partition consists only of numbers that are powers of two. Each step reduces a 2^{k+1} to 2^k and 2^k . Thus 16 is converted to 8 and 8.

Reference: TAOCP 4, section 7.2.1.5, problem 64

Examples

```

>>> from sympy.utilities.iterables import binary_partitions
>>> for i in binary_partitions(5):
...     print(i)
...
[4, 1]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]

```

`sympy.utilities.iterables.bracelets(n, k)`

Wrapper to necklaces to return a free (unrestricted) necklace.

`sympy.utilities.iterables.capture(func)`

Return the printed output of `func()`.

func should be a function without arguments that produces output with print statements.

```

>>> from sympy.utilities.iterables import capture
>>> from sympy import pprint
>>> from sympy.abc import x

```

(continues on next page)

(continued from previous page)

```
>>> def foo():
...     print('hello world!')
...
>>> 'hello' in capture(foo) # foo, not foo()
True
>>> capture(lambda: pprint(2/x))
'2\n-\nx\n'
```

`sympy.utilities.iterables.common_prefix(*seqs)`

Return the subsequence that is a common start of sequences in *seqs*.

```
>>> from sympy.utilities.iterables import common_prefix
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([1, 2, 3], [1, 2, 5])
[1, 2]
>>> common_prefix([1, 2, 3], [1, 3, 5])
[1]
```

`sympy.utilities.iterables.common_suffix(*seqs)`

Return the subsequence that is a common ending of sequences in *seqs*.

```
>>> from sympy.utilities.iterables import common_suffix
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([1, 2, 3], [9, 2, 3])
[2, 3]
>>> common_suffix([1, 2, 3], [9, 7, 3])
[3]
```

`sympy.utilities.iterables.dict_merge(*dicts)`

Merge dictionaries into a single dictionary.

`sympy.utilities.iterables.filter_symbols(iterator, exclude)`

Only yield elements from *iterator* that do not occur in *exclude*.

Parameters *iterator* : iterable

iterator to take elements from

exclude : iterable

elements to exclude

Returns *iterator* : iterator

filtered iterator

`sympy.utilities.iterables.flatten(iterable, levels=None, cls=None)`

Recursively denest iterable containers.

```
>>> from sympy.utilities.iterables import flatten
```

```
>>> flatten([1, 2, 3])
[1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
>>> flatten([1, 2, [3]])
[1, 2, 3]
>>> flatten([1, [2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> flatten([1.0, 2, (1, None)])
[1.0, 2, 1, None]
```

If you want to denest only a specified number of levels of nested containers, then set `levels` flag to the desired number of levels:

```
>>> ls = [[(-2, -1), (1, 2)], [(0, 0)]]
```

```
>>> flatten(ls, levels=1)
[(-2, -1), (1, 2), (0, 0)]
```

If `cls` argument is specified, it will only flatten instances of that class, for example:

```
>>> from sympy.core import Basic
>>> class MyOp(Basic):
...     pass
...
>>> flatten([MyOp(1, MyOp(2, 3))], cls=MyOp)
[1, 2, 3]
```

adapted from http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks

`sympy.utilities.iterables.generate_bell(n)`

Return permutations of $[0, 1, \dots, n-1]$ such that each permutation differs from the last by the exchange of a single pair of neighbors. The $n!$ permutations are returned as an iterator. In order to obtain the next permutation from a random starting permutation, use the `next_trotterjohnson` method of the `Permutation` class (which generates the same sequence in a different manner).

See also:

`sympy.combinatorics.Permutation.next_trotterjohnson`

References

- http://en.wikipedia.org/wiki/Method_ringing
- <http://stackoverflow.com/questions/4856615/recursive-permutation/4857018>
- <http://programminggeeks.com/bell-algorithm-for-permutation/>
- http://en.wikipedia.org/wiki/Steinhaus%E2%80%93Johnson%E2%80%93Trotter_algorithm
- Generating involutions, derangements, and relatives by ECO Vincent Vajnovszki, DMTCS vol 1 issue 12, 2010

Examples

```
>>> from itertools import permutations
>>> from sympy.utilities.iterables import generate_bell
>>> from sympy import zeros, Matrix
```

This is the sort of permutation used in the ringing of physical bells, and does not produce permutations in lexicographical order. Rather, the permutations differ from each other by exactly one inversion, and the position at which the swapping occurs varies periodically in a simple fashion. Consider the first few permutations of 4 elements generated by `permutations` and `generate_bell`:

```
>>> list(permutations(range(4))[:5])
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 2, 1, 3), (0, 2, 3, 1), (0, 3, 1, 2)]
>>> list(generate_bell(4)[:5])
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 3, 1, 2), (3, 0, 1, 2), (3, 0, 2, 1)]
```

Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each “bell” permutation always has only two elements out of place relative to the previous permutation (and so the signature (+/-1) of a permutation is opposite of the signature of the previous permutation).

How the position of inversion varies across the elements can be seen by tracing out where the largest number appears in the permutations:

```
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
...     m[:, i] = Matrix([j - 3 for j in list(p)]) # make largest zero
>>> m.print_nonzero('X')
[XXX XXXXXX XXXXXX XXX]
[XX XX XXXX XX XXXX XX XX]
[X XXXX XX XXXX XX XXXX X]
[ XXXXXX XXXXXX XXXXXX ]
```

`sympy.utilities.iterables.generate_derangements(perm)`

Routine to generate unique derangements.

TODO: This will be rewritten to use the ECO operator approach once the permutations branch is in master.

See also:

`sympy.functions.combinatorial.factorials.subfactorial` (page 466)

Examples

```
>>> from sympy.utilities.iterables import generate_derangements
>>> list(generate_derangements([0, 1, 2]))
[[1, 2, 0], [2, 0, 1]]
>>> list(generate_derangements([0, 1, 2, 3]))
[[1, 0, 3, 2], [1, 2, 3, 0], [1, 3, 0, 2], [2, 0, 3, 1], [2, 3, 0, 1], [2, 3, 1, 0],
 [3, 0, 1, 2], [3, 2, 0, 1], [3, 2, 1, 0]]
>>> list(generate_derangements([0, 1, 1]))
[]
```

`sympy.utilities.iterables.generate_involutions(n)`

Generates involutions.

An involution is a permutation that when multiplied by itself equals the identity permutation. In this implementation the involutions are generated using Fixed Points.

Alternatively, an involution can be considered as a permutation that does not contain any cycles with a length that is greater than two.

Reference: <http://mathworld.wolfram.com/PermutationInvolution.html>

Examples

```
>>> from sympy.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
>>> len(list(generate_involutions(4)))
10
```

`sympy.utilities.iterables.generate_oriented_forest(n)`

This algorithm generates oriented forests.

An oriented graph is a directed graph having no symmetric pair of directed edges. A forest is an acyclic graph, i.e., it has no cycles. A forest can also be described as a disjoint union of trees, which are graphs in which any two vertices are connected by exactly one simple path.

Reference: [1] T. Beyer and S.M. Hedetniemi: constant time generation of rooted trees, SIAM J. Computing Vol. 9, No. 4, November 1980 [2] <http://stackoverflow.com/questions/1633833/oriented-forest-taocp-algorithm-in-python>

Examples

```
>>> from sympy.utilities.iterables import generate_oriented_forest
>>> list(generate_oriented_forest(4))
[[0, 1, 2, 3], [0, 1, 2, 2], [0, 1, 2, 1], [0, 1, 2, 0], [0, 1, 1, 1], [0, 1, 1, 0], [0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 0]]
```

`sympy.utilities.iterables.group(seq, multiple=True)`

Splits a sequence into a list of lists of equal, adjacent elements.

See also:

`multiset` (page 1425)

Examples

```
>>> from sympy.utilities.iterables import group
```

```
>>> group([1, 1, 1, 2, 2, 3])
[[1, 1, 1], [2, 2], [3]]
>>> group([1, 1, 1, 2, 2, 3], multiple=False)
[(1, 3), (2, 2), (3, 1)]
>>> group([1, 1, 3, 2, 2, 1], multiple=False)
[(1, 2), (3, 1), (2, 2), (1, 1)]
```

`sympy.utilities.iterables.has_dups(seq)`

Return True if there are any duplicate elements in seq.

Examples

```
>>> from sympy.utilities.iterables import has_dups
>>> from sympy import Dict, Set
```

```
>>> has_dups((1, 2, 1))
True
>>> has_dups(range(3))
False
>>> all(has_dups(c) is False for c in (set(), Set(), dict(), Dict()))
True
```

`sympy.utilities.iterables.has_variety(seq)`
Return True if there are any different elements in seq.

Examples

```
>>> from sympy.utilities.iterables import has_variety
```

```
>>> has_variety((1, 2, 1))
True
>>> has_variety((1, 1, 1))
False
```

`sympy.utilities.iterables.ibin(n, bits=0, str=False)`
Return a list of length `bits` corresponding to the binary value of `n` with small bits to the right (last). If `bits` is omitted, the length will be the number required to represent `n`. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.
If a sequence of all bits-length lists starting from `[0, 0, ..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for `bits`, e.g. `'all'`.
If the bit string is desired pass `str=True`.

Examples

```
>>> from sympy.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
>>> ibin(2, 4)[::-1]
[0, 1, 0, 0]
```

If all lists corresponding to 0 to $2^{**n} - 1$, pass a non-integer for `bits`:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use `str=True`:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
```

(continues on next page)

(continued from previous page)

```
'0001111011'
>>> ibin(n, bits, str=True)[::-1] # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

`sympy.utilities.iterables.interactive_traversal(expr)`
 Traverse a tree asking a user which branch to choose.

`sympy.utilities.iterables.kbins(l, k, ordered=None)`
 Return sequence `l` partitioned into `k` bins.

See also:

`partitions` (page 1429), `multiset_partitions` (page 1425)

Examples

```
>>> from sympy.utilities.iterables import kbins
```

The default is to give the items in the same order, but grouped into `k` partitions without any reordering:

```
>>> from __future__ import print_function
>>> for p in kbins(list(range(5)), 2):
...     print(p)
...
[[0], [1, 2, 3, 4]]
[[0, 1], [2, 3, 4]]
[[0, 1, 2], [3, 4]]
[[0, 1, 2, 3], [4]]
```

The `ordered` flag is either `None` (to give the simple partition of the elements) or is a 2 digit integer indicating whether the order of the bins and the order of the items in the bins matters. Given:

```
A = [[0], [1, 2]]
B = [[1, 2], [0]]
C = [[2, 1], [0]]
D = [[0], [2, 1]]
```

the following values for `ordered` have the shown meanings:

```
00 means A == B == C == D
01 means A == B
10 means A == D
11 means A == A
```

```
>>> for ordered in [None, 0, 1, 10, 11]:
...     print('ordered = %s' % ordered)
...     for p in kbins(list(range(3)), 2, ordered=ordered):
...         print('      %s' % p)
...
ordered = None
[[0], [1, 2]]
```

(continues on next page)

(continued from previous page)

```

[[0, 1], [2]]
ordered = 0
[[0, 1], [2]]
[[0, 2], [1]]
[[0], [1, 2]]
ordered = 1
[[0], [1, 2]]
[[0], [2, 1]]
[[1], [0, 2]]
[[1], [2, 0]]
[[2], [0, 1]]
[[2], [1, 0]]
ordered = 10
[[0, 1], [2]]
[[2], [0, 1]]
[[0, 2], [1]]
[[1], [0, 2]]
[[0], [1, 2]]
[[1, 2], [0]]
ordered = 11
[[0], [1, 2]]
[[0, 1], [2]]
[[0], [2, 1]]
[[0, 2], [1]]
[[1], [0, 2]]
[[1, 0], [2]]
[[1], [2, 0]]
[[1, 2], [0]]
[[2], [0, 1]]
[[2, 0], [1]]
[[2], [1, 0]]
[[2, 1], [0]]

```

`sympy.utilities.iterables.minlex(seq, directed=True, is_set=False, small=None)`

Return a tuple where the smallest element appears first; if `directed` is `True` (default) then the order is preserved, otherwise the sequence will be reversed if that gives a smaller ordering.

If every element appears only once then `is_set` can be set to `True` for more efficient processing.

If the smallest element is known at the time of calling, it can be passed and the calculation of the smallest element will be omitted.

Examples

```

>>> from sympy.combinatorics.polyhedron import minlex
>>> minlex((1, 2, 0))
(0, 1, 2)
>>> minlex((1, 0, 2))
(0, 2, 1)
>>> minlex((1, 0, 2), directed=False)
(0, 1, 2)

```

```
>>> minlex('11010011000', directed=True)
'00011010011'
>>> minlex('11010011000', directed=False)
'00011001011'
```

`sympy.utilities.iterables.multiset(seq)`

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

See also:

[group](#) (page 1421)

Examples

```
>>> from sympy.utilities.iterables import multiset
>>> multiset('mississippi')
{'i': 4, 'm': 1, 'p': 2, 's': 4}
```

`sympy.utilities.iterables.multiset_combinations(m, n, g=None)`

Return the unique combinations of size `n` from multiset `m`.

Examples

```
>>> from sympy.utilities.iterables import multiset_combinations
>>> from itertools import combinations
>>> [''.join(i) for i in multiset_combinations('baby', 3)]
['abb', 'aby', 'bby']
```

```
>>> def count(f, s): return len(list(f(s, 3)))
```

The number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

```
>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)
```

`sympy.utilities.iterables.multiset_partitions(multiset, m=None)`

Return unique partitions of the given multiset (in list form). If `m` is `None`, all multisets will be returned, otherwise only partitions with `m` parts will be returned.

If multiset is an integer, a range `[0, 1, ..., multiset - 1]` will be supplied.

See also:

[partitions](#) (page 1429), [sympy.combinatorics.partitions.Partition](#) (page 202), [sympy.combinatorics.partitions.IntegerPartition](#) (page 204), [sympy.functions.combinatorial.numbers.nT](#)

Notes

When all the elements are the same in the multiset, the order of the returned partitions is determined by the partitions routine. If one is counting partitions then it is better to use the nT function.

Examples

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> list(multiset_partitions([1, 2, 3, 4], 2))
[[[1, 2, 3], [4]], [[1, 2, 4], [3]], [[1, 2], [3, 4]],
 [[1, 3, 4], [2]], [[1, 3], [2, 4]], [[1, 4], [2, 3]],
 [[1], [2, 3, 4]]]
>>> list(multiset_partitions([1, 2, 3, 4], 1))
[[[1, 2, 3, 4]]]
```

Only unique partitions are returned and these will be returned in a canonical order regardless of the order of the input:

```
>>> a = [1, 2, 2, 1]
>>> ans = list(multiset_partitions(a, 2))
>>> a.sort()
>>> list(multiset_partitions(a, 2)) == ans
True
>>> a = range(3, 1, -1)
>>> (list(multiset_partitions(a)) ==
...  list(multiset_partitions(sorted(a))))
True
```

If m is omitted then all partitions will be returned:

```
>>> list(multiset_partitions([1, 1, 2]))
[[[1, 1, 2]], [[1, 1], [2]], [[1, 2], [1]], [[1], [1], [2]]]
>>> list(multiset_partitions([1]*3))
[[[1, 1, 1]], [[1], [1, 1]], [[1], [1], [1]]]
```

Counting

The number of partitions of a set is given by the bell number:

```
>>> from sympy import bell
>>> len(list(multiset_partitions(5))) == bell(5) == 52
True
```

The number of partitions of length k from a set of size n is given by the Stirling Number of the 2nd kind:

```
>>> def S2(n, k):
...     from sympy import Dummy, binomial, factorial, Sum
...     if k > n:
...         return 0
...     j = Dummy()
...     arg = (-1)**(k-j)*j**n*binomial(k,j)
```

(continues on next page)

(continued from previous page)

```
...     return 1/factorial(k)*Sum(arg,(j,0,k)).doit()
...
>>> S2(5, 2) == len(list(multiset_partitions(5, 2))) == 15
True
```

These comments on counting apply to sets, not multisets.

`sympy.utilities.iterables.multiset_permutations(m, size=None, g=None)`
Return the unique permutations of multiset `m`.

Examples

```
>>> from sympy.utilities.iterables import multiset_permutations
>>> from sympy import factorial
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

`sympy.utilities.iterables.necklaces(n, k, free=False)`

A routine to generate necklaces that may (`free=True`) or may not (`free=False`) be turned over to be viewed. The “necklaces” returned are comprised of `n` integers (beads) with `k` different values (colors). Only unique necklaces are returned.

References

<http://mathworld.wolfram.com/Necklace.html>

Examples

```
>>> from sympy.utilities.iterables import necklaces, bracelets
>>> def show(s, i):
...     return ''.join(s[j] for j in i)
```

The “unrestricted necklace” is sometimes also referred to as a “bracelet” (an object that can be turned over, a sequence that can be reversed) and the term “necklace” is used to imply a sequence that cannot be reversed. So `ACB == ABC` for a bracelet (rotate and reverse) while the two are different for a necklace since rotation alone cannot make the two sequences the same.

(mnemonic: Bracelets can be viewed Backwards, but Not Necklaces.)

```
>>> B = [show('ABC', i) for i in bracelets(3, 3)]
>>> N = [show('ABC', i) for i in necklaces(3, 3)]
>>> set(N) - set(B)
{'ACB'}
```

```
>>> list(necklaces(4, 2))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 1),
 (0, 1, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1)]
```

```
>>> [show('.o', i) for i in bracelets(4, 2)]
['....', '...o', '..oo', '.o.o', '.ooo', 'oooo']
```

`sympy.utilities.iterables.numbered_symbols(prefix='x', cls=None, start=0, exclude=[], *args, **assumptions)`

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in *exclude*.

Parameters *prefix* : str, optional

The prefix to use. By default, this function will generate symbols of the form “x0”, “x1”, etc.

cls : class, optional

The class to use. By default, it uses Symbol, but you can also use Wild or Dummy.

start : int, optional

The start number. By default, it is 0.

Returns *sym* : Symbol

The subscripted symbols.

`sympy.utilities.iterables.ordered_partitions(n, m=None, sort=True)`

Generates ordered partitions of integer *n*.

Parameters “*m*” : integer (default gives partitions of all sizes) else only

those with size *m*. In addition, if *m* is not None then partitions are generated in place (see examples).

“*sort*” : bool (default True) controls whether partitions are

returned in sorted order when *m* is not None; when False, the partitions are returned as fast as possible with elements sorted, but when *m* | *n* the partitions will not be in ascending lexicographical order.

References

[R557] (page 1872), [R558] (page 1872)

Examples

```
>>> from sympy.utilities.iterables import ordered_partitions
```

All partitions of 5 in ascending lexicographical:

```
>>> for p in ordered_partitions(5):
...     print(p)
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 3]
[1, 2, 2]
[1, 4]
[2, 3]
[5]
```

Only partitions of 5 with two parts:

```
>>> for p in ordered_partitions(5, 2):
...     print(p)
[1, 4]
[2, 3]
```

When `m` is given, a given list objects will be used more than once for speed reasons so you will not see the correct partitions unless you make a copy of each as it is generated:

```
>>> [p for p in ordered_partitions(7, 3)]
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [2, 2, 2]]
>>> [list(p) for p in ordered_partitions(7, 3)]
[[1, 1, 5], [1, 2, 4], [1, 3, 3], [2, 2, 3]]
```

When `n` is a multiple of `m`, the elements are still sorted but the partitions themselves will be unordered if `sort` is `False`; the default is to return them in ascending lexicographical order.

```
>>> for p in ordered_partitions(6, 2):
...     print(p)
[1, 5]
[2, 4]
[3, 3]
```

But if speed is more important than ordering, `sort` can be set to `False`:

```
>>> for p in ordered_partitions(6, 2, sort=False):
...     print(p)
[1, 5]
[3, 3]
[2, 4]
```

`sympy.utilities.iterables.partitions(n, m=None, k=None, size=False)`
Generate all partitions of positive integer, `n`.

Parameters “`m`” : integer (default gives partitions of all sizes)

limits number of parts in partition (mnemonic: `m`, maximum parts)

“`k`” : integer (default gives partitions number from 1 through `n`)

limits the numbers that are kept in the partition (mnemonic: `k`, keys)

“`size`” : bool (default `False`, only partition is returned)

when `True` then `(M, P)` is returned where `M` is the sum of the multiplicities and `P` is the generated partition.

Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is {4: 1}, “4: one of them”.

See also:

[sympy.combinatorics.partitions.Partition](#) (page 202), [sympy.combinatorics.partitions.IntegerPartition](#) (page 204)

Examples

```
>>> from sympy.utilities.iterables import partitions
```

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```
>>> for p in partitions(6, k=2):
...     print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m (default value, None, gives partitions from 1 through n):

```
>>> for p in partitions(6, m=2):
...     print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

Note that the `_same_` dictionary object is returned each time. This is for speed: generating each partition goes quickly, taking constant time, independent of n.

```
>>> [p for p in partitions(6, k=2)]
[{1: 6}, {1: 6}, {1: 6}, {1: 6}]
```

If you want to build a list of the returned dictionaries then make a copy of them:

```
>>> [p.copy() for p in partitions(6, k=2)]
[{2: 3}, {1: 2, 2: 2}, {1: 4, 2: 1}, {1: 6}]
>>> [(M, p.copy()) for M, p in partitions(6, k=2, size=True)]
[(3, {2: 3}), (4, {1: 2, 2: 2}), (5, {1: 4, 2: 1}), (6, {1: 6})]
```

Reference: modified from Tim Peter's version to allow for k and m values:
code.activestate.com/recipes/218332-generator-for-integer-partitions/

`sympy.utilities.iterables.permute_signs(t)`

Return iterator in which the signs of non-zero elements of t are permuted.

Examples

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2)]
```

`sympy.utilities.iterables.postfixes(seq)`

Generate all postfixes of a sequence.

Examples

```
>>> from sympy.utilities.iterables import postfixes
```

```
>>> list(postfixes([1,2,3,4]))
[[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

`sympy.utilities.iterables.postorder_traversal(node, keys=None)`

Do a postorder traversal of a tree.

This generator recursively yields nodes that it has visited in a postorder fashion. That is, it descends through the tree depth-first to yield all of a node's children's postorder traversal before yielding the node itself.

Parameters `node` : sympy expression

The expression to traverse.

keys : (default None) sort key(s)

The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to `ordered()` as the only key(s) to use to sort the arguments; if key is simply True then the default keys of `ordered` will be used (node count and `default_sort_key`).

Yields `subtree` : sympy expression

All of the subtrees in the tree.

Examples

```
>>> from sympy.utilities.iterables import postorder_traversal
>>> from sympy.abc import w, x, y, z
```

The nodes are returned in the order that they are encountered unless key is given; simply passing `key=True` will guarantee that the traversal is unique.

```
>>> list(postorder_traversal(w + (x + y)*z))
[z, y, x, x + y, z*(x + y), w, w + z*(x + y)]
>>> list(postorder_traversal(w + (x + y)*z, keys=True))
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

`sympy.utilities.iterables.prefixes(seq)`

Generate all prefixes of a sequence.

Examples

```
>>> from sympy.utilities.iterables import prefixes
```

```
>>> list(prefixes([1,2,3,4]))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

`sympy.utilities.iterables.reshape(seq, how)`

Reshape the sequence according to the template in `how`.

Examples

```
>>> from sympy.utilities import reshape
>>> seq = list(range(1, 9))
```

```
>>> reshape(seq, [4]) # lists of 4
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
>>> reshape(seq, (4,)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, 2)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, [2])) # (i, i, [i, i])
[(1, 2, [3, 4]), (5, 6, [7, 8])]
```

```
>>> reshape(seq, ((2,), [2])) # etc....
[((1, 2), [3, 4]), ((5, 6), [7, 8])]
```

```
>>> reshape(seq, (1, [2], 1))
[(1, [2, 3], 4), (5, [6, 7], 8)]
```

```
>>> reshape(tuple(seq), ([[1], 1, (2,)]))
([[[1], 2, (3, 4)],], ([[5], 6, (7, 8)],])
```

```
>>> reshape(tuple(seq), ([1], 1, (2,)))
([[[1], 2, (3, 4)], ([5], 6, (7, 8))])
```

```
>>> reshape(list(range(12)), [2, [3], {2}, (1, (3,)), 1])
[[0, 1, [2, 3, 4], {5, 6}, (7, (8, 9, 10), 11)]]
```

`sympy.utilities.iterables.rotate_left(x, y)`
Left rotates a list `x` by the number of steps specified in `y`.

Examples

```
>>> from sympy.utilities.iterables import rotate_left
>>> a = [0, 1, 2]
>>> rotate_left(a, 1)
[1, 2, 0]
```

`sympy.utilities.iterables.rotate_right(x, y)`
Right rotates a list `x` by the number of steps specified in `y`.

Examples

```
>>> from sympy.utilities.iterables import rotate_right
>>> a = [0, 1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> rotate_right(a, 1)
[2, 0, 1]
```

`sympy.utilities.iterables.runs(seq, op=<built-in function gt>)`

Group the sequence into lists in which successive elements all compare the same with the comparison operator, `op`: `op(seq[i + 1], seq[i])` is True from all elements in a run.

Examples

```
>>> from sympy.utilities.iterables import runs
>>> from operator import ge
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2])
[[0, 1, 2], [2], [1, 4], [3], [2], [2]]
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2], op=ge)
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

`sympy.utilities.iterables.sift(seq, keyfunc, binary=False)`

Sift the sequence, `seq` according to `keyfunc`.

OUTPUT: When `binary` is False (default), the output is a dictionary where elements of `seq` are stored in a list keyed to the value of `keyfunc` for that element. If `binary` is True then a tuple with lists T and F are returned where T is a list containing elements of `seq` for which `keyfunc` was True and F containing those elements for which `keyfunc` was False; a `ValueError` is raised if the `keyfunc` is not binary.

See also:

`ordered`

Examples

```
>>> from sympy.utilities import sift
>>> from sympy.abc import x, y
>>> from sympy import sqrt, exp, pi, Tuple
```

```
>>> sift(range(5), lambda x: x % 2)
{0: [0, 2, 4], 1: [1, 3]}
```

`sift()` returns a `defaultdict()` object, so any key that has no matches will give `[]`.

```
>>> sift([x], lambda x: x.is_commutative)
{True: [x]}
>>> _[False]
[]
```

Sometimes you won't know how many keys you will get:

```
>>> sift([sqrt(x), exp(x), (y**x)**2],
...      lambda x: x.as_base_exp()[0])
{E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}
```

Sometimes you expect the results to be binary; the results can be unpacked by setting `binary` to True:

```
>>> sift(range(4), lambda x: x % 2, binary=True)
([1, 3], [0, 2])
>>> sift(Tuple(1, pi), lambda x: x.is_rational, binary=True)
([1], [pi])
```

A `ValueError` is raised if the predicate was not actually binary (which is a good test for the logic where sifting is used and binary results were expected):

```
>>> unknown = exp(1) - pi # the rationality of this is unknown
>>> args = Tuple(1, pi, unknown)
>>> sift(args, lambda x: x.is_rational, binary=True)
Traceback (most recent call last):
...
ValueError: keyfunc gave non-binary output
```

The non-binary sifting shows that there were 3 keys generated:

```
>>> set(sift(args, lambda x: x.is_rational).keys())
{None, False, True}
```

If you need to sort the sifted items it might be better to use `ordered` which can economically apply multiple sort keys to a sequence while sorting.

`sympy.utilities.iterables.signed_permutations(t)`

Return iterator in which the signs of non-zero elements of `t` and the order of the elements are permuted.

Examples

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2), (0, 2, 1),
(0, -2, 1), (0, 2, -1), (0, -2, -1), (1, 0, 2), (-1, 0, 2),
(1, 0, -2), (-1, 0, -2), (1, 2, 0), (-1, 2, 0), (1, -2, 0),
(-1, -2, 0), (2, 0, 1), (-2, 0, 1), (2, 0, -1), (-2, 0, -1),
(2, 1, 0), (-2, 1, 0), (2, -1, 0), (-2, -1, 0)]
```

`sympy.utilities.iterables.subsets(seq, k=None, repetition=False)`

Generates all k -subsets (combinations) from an n -element set, `seq`.

A k -subset of an n -element set is any subset of length exactly k . The number of k -subsets of an n -element set is given by $\text{binomial}(n, k)$, whereas there are 2^n subsets all together. If k is `None` then all 2^n subsets will be returned from shortest to longest.

Examples

```
>>> from sympy.utilities.iterables import subsets
```

`subsets(seq, k)` will return the $n!/k!(n-k)!$ k -subsets (combinations) without repetition, i.e. once an item has been removed, it can no longer be “taken”:

```
>>> list(subsets([1, 2], 2))
[(1, 2)]
>>> list(subsets([1, 2]))
```

(continues on next page)

(continued from previous page)

```
[(), (1,), (2,), (1, 2)]
>>> list(subsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
```

`subsets(seq, k, repetition=True)` will return the $(n - 1 + k)!/k!/(n - 1)!$ combinations with repetition:

```
>>> list(subsets([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(subsets([0, 1], 3, repetition=False))
[]
>>> list(subsets([0, 1], 3, repetition=True))
[(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)]
```

`sympy.utilities.iterables.take(iter, n)`

Return `n` items from `iter` iterator.

`sympy.utilities.iterables.topological_sort(graph, key=None)`

Topological sort of graph's vertices.

Parameters “`graph`”: `tuple[list, list[tuple[T, T]]]`

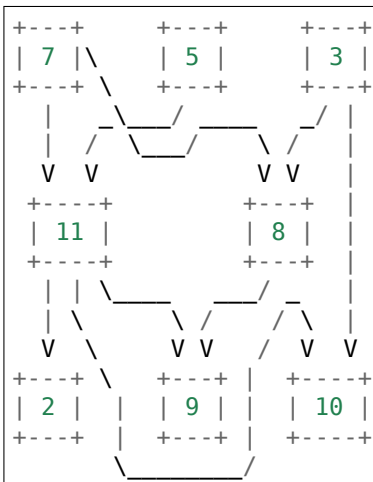
A tuple consisting of a list of vertices and a list of edges of a graph to be sorted topologically.

“`key`”: `callable[T]` (optional)

Ordering key for vertices on the same level. By default the natural (e.g. lexicographic) ordering is used (in this case the base type must implement ordering relations).

Examples

Consider a graph:



where vertices are integers. This graph can be encoded using elementary Python's data structures as follows:

```
>>> V = [2, 3, 5, 7, 8, 9, 10, 11]
>>> E = [(7, 11), (7, 8), (5, 11), (3, 8), (3, 10),
...      (11, 2), (11, 9), (11, 10), (8, 9)]
```

To compute a topological sort for graph (V, E) issue:

```
>>> from sympy.utilities.iterables import topological_sort

>>> topological_sort((V, E))
[3, 5, 7, 8, 11, 2, 9, 10]
```

If specific tie breaking approach is needed, use key parameter:

```
>>> topological_sort((V, E), key=lambda v: -v)
[7, 5, 11, 3, 10, 8, 9, 2]
```

Only acyclic graphs can be sorted. If the input graph has a cycle, then `ValueError` will be raised:

```
>>> topological_sort((V, E + [(10, 7)]))
Traceback (most recent call last):
...
ValueError: cycle detected
```

See also:

http://en.wikipedia.org/wiki/Topological_sorting

`sympy.utilities.iterables.unflatten(iter, n=2)`

Group iter into tuples of length n. Raise an error if the length of iter is not a multiple of n.

`sympy.utilities.iterables.uniq(seq, result=None)`

Yield unique elements from seq as an iterator. The second parameter result is used internally; it is not necessary to pass anything for this.

Examples

```
>>> from sympy.utilities.iterables import uniq
>>> dat = [1, 4, 1, 5, 4, 2, 1, 2]
>>> type(uniq(dat)) in (list, tuple)
False
```

```
>>> list(uniq(dat))
[1, 4, 5, 2]
>>> list(uniq(x for x in dat))
[1, 4, 5, 2]
>>> list(uniq([[1], [2, 1], [1]]))
[[1], [2, 1]]
```

`sympy.utilities.iterables.variations(seq, n, repetition=False)`

Returns a generator of the n-sized variations of seq (size N). repetition controls whether items in seq can appear more than once;

See also:

`sympy.core.compatibility.permutations`, `sympy.core.compatibility.product`

Examples

`variations(seq, n)` will return $N! / (N - n)!$ permutations without repetition of `seq`'s elements:

```
>>> from sympy.utilities.iterables import variations
>>> list(variations([1, 2], 2))
[(1, 2), (2, 1)]
```

`variations(seq, n, True)` will return the N^n permutations obtained by allowing repetition of elements:

```
>>> list(variations([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(variations([0, 1], 3, repetition=False))
[]
>>> list(variations([0, 1], 3, repetition=True))[:4]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)]
```

5.36.6 Lambdify

This module provides convenient functions to transform sympy expressions to lambda functions which can be used to calculate numerical values very fast.

`sympy.utilities.lambdify.implemented_function(symfunc, implementation)`

Add numerical implementation to function `symfunc`.

`symfunc` can be an `UndefinedFunction` instance, or a name string. In the latter case we create an `UndefinedFunction` instance with that name.

Be aware that this is a quick workaround, not a general method to create special symbolic functions. If you want to create a symbolic function to be used by all the machinery of SymPy you should subclass the `Function` class.

Parameters `symfunc` : str or `UndefinedFunction` instance

If str, then create new `UndefinedFunction` with this as name. If `symfunc` is an `Undefined` function, create a new function with the same name and the implemented function attached.

implementation : callable

numerical implementation to be called by `evalf()` or `lambdify`

Returns `afunc` : `sympy.FunctionClass` instance

function with attached implementation

Examples

```
>>> from sympy.abc import x
>>> from sympy.utilities.lambdify import lambdify, implemented_function
>>> from sympy import Function
```

(continues on next page)

(continued from previous page)

```
>>> f = implemented_function('f', lambda x: x+1)
>>> lam_f = lambdify(x, f(x))
>>> lam_f(4)
5
```

`sympy.utilities.lambdify.lambdastr(args, expr, printer=None, dummify=False)`
Returns a string that can be evaluated to a lambda function.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.lambdify import lambdastr
>>> lambdastr(x, x**2)
'lambda x: (x**2)'
>>> lambdastr((x,y,z), [z,y,x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, `lambdastr` will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))(_0,_1[0],_1[1])'
```

`sympy.utilities.lambdify.lambdify(args, expr, modules=None, printer=None, use_ims=True, dummify=False)`

Returns an anonymous function for fast calculation of numerical values.

If not specified differently by the user, `modules` defaults to `["numpy"]` if NumPy is installed, and `["math", "mpmath", "sympy"]` if it isn't, that is, SymPy functions are replaced as far as possible by either numpy functions if available, and Python's standard library `math`, or `mpmath` functions otherwise. To change this behavior, the "modules" argument can be used. It accepts:

- the strings "math", "mpmath", "numpy", "numexpr", "sympy", "tensorflow"
- any modules (e.g. `math`)
- dictionaries that map names of sympy functions to arbitrary functions
- lists that contain a mix of the arguments above, with higher priority given to entries appearing first.

Warning: Note that this function uses `eval`, and thus shouldn't be used on unsanitized input.

Arguments in the provided expression that are not valid Python identifiers are substituted with dummy symbols. This allows for applied functions (e.g. `f(t)`) to be supplied as arguments. Call the function with `dummify=True` to replace all arguments with dummy symbols (if `args` is not a string) - for example, to ensure that the arguments do not redefine any built-in names.

For functions involving large array calculations, `numexpr` can provide a significant speedup over `numpy`. Please note that the available functions for `numexpr` are more limited than `numpy` but can be expanded with `implemented_function` and user defined

subclasses of Function. If specified, numexpr may be the only option in modules. The official list of numexpr functions can be found at: <https://github.com/pydata/numexpr#supported-functions>

In previous releases lambdify replaced Matrix with numpy.matrix by default. As of release 1.0 numpy.array is the default. To get the old default behavior you must pass in [{'ImmutableDenseMatrix': numpy.matrix}, 'numpy'] to the modules kwarg.

```
>>> from sympy import lambdify, Matrix
>>> from sympy.abc import x, y
>>> import numpy
>>> array2mat = [{'ImmutableDenseMatrix': numpy.matrix}, 'numpy']
>>> f = lambdify((x, y), Matrix([x, y]), modules=array2mat)
>>> f(1, 2)
matrix([[1],
        [2]])
```

Examples

```
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import sqrt, sin, Matrix
>>> from sympy import Function
>>> from sympy.abc import w, x, y, z
```

```
>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1,2,3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='sympy')
>>> row(1, 2)
Matrix([[1, 3]])
```

Tuple arguments are handled and the lambdified function should be called with the same type of arguments as were used to create the function.:

```
>>> f = lambdify((x, (y, z)), x + y)
>>> f(1, (2, 4))
3
```

A more robust way of handling this is to always work with flattened arguments:

```
>>> from sympy.utilities.iterables import flatten
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```

Functions present in *expr* can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. Usually you attach this using the `implemented_function` factory:

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

`lambdify` always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_ims` input parameter is `False`.

Usage with Tensorflow module:

```
>>> import tensorflow as tf
>>> f = Max(x, sin(x))
>>> func = lambdify(x, f, 'tensorflow')
>>> result = func(tf.constant(1.0))
>>> result # a tf.Tensor representing the result of the calculation
<tf.Tensor 'Maximum:0' shape=() dtype=float32>
>>> sess = tf.Session()
>>> sess.run(result) # compute result
1.0
>>> var = tf.Variable(1.0)
>>> sess.run(tf.global_variables_initializer())
>>> sess.run(func(var)) # also works for tf.Variable and tf.Placeholder
1.0
>>> tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]]) # works with any shape tensor
>>> sess.run(func(tensor))
array([[ 1.,  2.],
       [ 3.,  4.]], dtype=float32)
```

Usage

1. Use one of the provided modules:

```
>>> from sympy import sin, tan, gamma
>>> from sympy.abc import x, y
>>> f = lambdify(x, sin(x), "math")
```

Attention: Functions that are not in the `math` module will throw a name error when the function definition is evaluated! So this would be better:

```
>>> f = lambdify(x, sin(x)*gamma(x), ("math", "mpmath", "sympy"))
```

2. Use some other module:

```
>>> import numpy
>>> f = lambdify((x,y), tan(x*y), numpy)
```

Attention: There are naming differences between `numpy` and `sympy`. So if you simply take the `numpy` module, e.g. `sympy.atan` will not be translated to `numpy.arctan`. Use the modified module instead by passing the string `"numpy"`:

```
>>> f = lambdify((x,y), tan(x*y), "numpy")
>>> f(1, 2)
-2.18503986326
>>> from numpy import array
>>> f(array([1, 2, 3]), array([2, 3, 5]))
[-2.18503986 -0.29100619 -0.8559934 ]
```

In the above examples, the generated functions can accept scalar values or numpy arrays as arguments. However, in some cases the generated function relies on the input being a numpy array:

```
>>> from sympy import Piecewise
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "numpy")
>>> f(array([-1, 0, 1, 2]))
[-1.  0.  1.  0.5]
>>> f(0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

In such cases, the input should be wrapped in a numpy array: `>>> float(f(array([0]))) 0.0`

Or if numpy functionality is not required another module can be used: `>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "math") >>> f(0) 0`

3. Use a dictionary defining custom functions:

```
>>> def my_cool_function(x): return 'sin(%s) is cool' % x
>>> myfuncs = {"sin": my_cool_function}
>>> f = lambdify(x, sin(x), myfuncs); f(1)
'sin(1) is cool'
```

5.36.7 Memoization

`sympy.utilities.memoization.assoc_recurrence_memo(base_seq)`

Memo decorator for associated sequences defined by recurrence starting from base

`base_seq(n)` - callable to get base sequence elements

XXX works only for $P_n = \text{base\_seq}(0)$ cases XXX works only for $m \leq n$ cases

`sympy.utilities.memoization.recurrence_memo(initial)`

Memo decorator for sequences defined by recurrence

See usage examples e.g. in the `specfun/combinatorial` module

5.36.8 Miscellaneous

Miscellaneous stuff that doesn't really fit anywhere else.

`sympy.utilities.misc.debug(*args)`

Print `*args` if `SYMPY_DEBUG` is True, else do nothing.

`sympy.utilities.misc.debug_decorator(func)`

If `SYMPY_DEBUG` is True, it will print a nice execution tree with arguments and results of all decorated functions, else do nothing.

`sympy.utilities.misc.filldedent(s, w=70)`

Strips leading and trailing empty lines from a copy of *s*, then dedents, fills and returns it.

Empty line stripping serves to deal with docstrings like this one that start with a newline after the initial triple quote, inserting an empty line at the beginning of the string.

`sympy.utilities.misc.find_executable(executable, path=None)`

Try to find 'executable' in the directories listed in 'path' (a string listing directories separated by 'os.pathsep'; defaults to os.environ['PATH']). Returns the complete filename or None if not found

`sympy.utilities.misc.func_name(x, short=False)`

Return function name of *x* (if defined) else the *type(x)*. If short is True and there is a shorter alias for the result, return the alias.

See also:

[sympy.core.compatibility](#) (page 198)

Examples

```
>>> from sympy.utilities.misc import func_name
>>> from sympy.abc import x
>>> func_name(x < 1)
'StrictLessThan'
>>> func_name(x < 1, short=True)
'Lt'
```

`sympy.utilities.misc.rawlines(s)`

Return a cut-and-pastable string that, when printed, is equivalent to the input. The string returned is formatted so it can be indented nicely within tests; in some cases it is wrapped in the dedent function which has to be imported from textwrap.

Examples

Note: because there are characters in the examples below that need to be escaped because they are themselves within a triple quoted docstring, expressions below look more complicated than they would be if they were printed in an interpreter window.

```
>>> from sympy.utilities.misc import rawlines
>>> from sympy import TableForm
>>> s = str(TableForm([[1, 10]], headings=(None, ['a', 'bee'])))
>>> print(rawlines(s))
(
    'a bee\n'
    '-----\n'
    '1 10 '
)
>>> print(rawlines(''''this
... that'''))
dedent('''\
    this
    that''')
```



```
>>> print(rawlines(''this
... that
... '''))
dedent(''\
    this
    that
    ''')
```

```
>>> s = """this
... is a triple '''
... """
>>> print(rawlines(s))
dedent("""\
    this
    is a triple '''
    """)
```

```
>>> print(rawlines(''this
... that
... '''))
(
    'this\n'
    'that\n'
    ,
)
```

`sympy.utilities.misc.replace(string, *reps)`

Return string with all keys in `reps` replaced with their corresponding values, longer strings first, irrespective of the order they are given. `reps` may be passed as tuples or a single mapping.

References

[R559] (page 1872)

Examples

```
>>> from sympy.utilities.misc import replace
>>> replace('foo', {'oo': 'ar', 'f': 'b'})
'bar'
>>> replace("spamham sha", ("spam", "eggs"), ("sha", "md5"))
'eggsham md5'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap (i.e. are the same length and have some identical sequence at the beginning/end):

```
>>> reps = [
...     ('ab', 'x'),
...     ('bc', 'y')]
>>> replace('abc', *reps) in ('xc', 'ay')
True
```

`sympy.utilities.misc.translate(s, a, b=None, c=None)`

Return `s` where characters have been replaced or deleted.

Examples

```
>>> from sympy.utilities.misc import translate
>>> from sympy.core.compatibility import unichr
>>> abc = 'abc'
>>> translate(abc, None, 'a')
'bc'
>>> translate(abc, {'a': 'x'}, 'c')
'xb'
>>> translate(abc, {'abc': 'x', 'a': 'y'})
'x'
```

```
>>> translate('abcd', 'ac', 'AC', 'd')
'AbC'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap are the same length and have some identical sequences at the beginning/end:

```
>>> translate(abc, {'ab': 'x', 'bc': 'y'}) in ('xc', 'ay')
True
```

Syntax

translate(s, None, deletechars): all characters in deletechars are deleted

translate(s, map [,deletechars]): all characters in deletechars (if provided) are deleted then the replacements defined by map are made; if the keys of map are strings then the longer ones are handled first. Multicharacter deletions should have a value of "".

translate(s, oldchars, newchars, deletechars) all characters in deletechars are deleted then each character in oldchars is replaced with the corresponding character in newchars

5.36.9 PKGDATA

pkgdata is a simple, extensible way for a package to acquire data file resources.

The getResource function is equivalent to the standard idioms, such as the following minimal implementation:

```
import sys, os

def getResource(identifier, pkgname=__name__):
    pkgpath = os.path.dirname(sys.modules[pkgname].__file__)
    path = os.path.join(pkgpath, identifier)
    return open(os.path.normpath(path), mode='rb')
```

When a `__loader__` is present on the module given by `__name__`, it will defer getResource to its `get_data` implementation and return it as a file-like object (such as StringIO).

`sympy.utilities.pkgdata.get_resource(identifier, pkgname='sympy.utilities.pkgdata')`
Acquire a readable object for a given package name and identifier. An IOError will be raised if the resource can not be found.

For example:

```
mydata = get_resource('mypkgdata.jpg').read()
```

Note that the package name must be fully qualified, if given, such that it would be found in `sys.modules`.

In some cases, `getResource` will return a real file object. In that case, it may be useful to use its `name` attribute to get the path rather than use it as a file-like object. For example, you may be handing data off to a C API.

5.36.10 pytest

pytest hacks to support XFAIL/XPASS

`sympy.utilities.pytest.SKIP(reason)`

Similar to `skip()`, but this is a decorator.

`sympy.utilities.pytest.raises(expectedException, code=None)`

Tests that code raises the exception `expectedException`.

`code` may be a callable, such as a lambda expression or function name.

If `code` is not given or `None`, `raises` will return a context manager for use in `with` statements; the code to execute then comes from the scope of the `with`.

`raises()` does nothing if the callable raises the expected exception, otherwise it raises an `AssertionError`.

Examples

```
>>> from sympy.utilities.pytest import raises
```

```
>>> raises(ZeroDivisionError, lambda: 1/0)
>>> raises(ZeroDivisionError, lambda: 1/2)
Traceback (most recent call last):
...
AssertionError: DID NOT RAISE
```

```
>>> with raises(ZeroDivisionError):
...     n = 1/0
>>> with raises(ZeroDivisionError):
...     n = 1/2
Traceback (most recent call last):
...
AssertionError: DID NOT RAISE
```

Note that you cannot test multiple statements via `with raises`:

```
>>> with raises(ZeroDivisionError):
...     n = 1/0      # will execute and raise, aborting the "with"
...     n = 9999/0  # never executed
```

This is just what `with` is supposed to do: abort the contained statement sequence at the first exception and let the context manager deal with the exception.

To test multiple statements, you'll need a separate `with` for each:

```
>>> with raises(ZeroDivisionError):  
...     n = 1/0      # will execute and raise  
>>> with raises(ZeroDivisionError):  
...     n = 9999/0 # will also execute and raise
```

5.36.11 Randomised Testing

Helpers for randomized testing

`sympy.utilities.randtest.random_complex_number(a=2, b=-1, c=3, d=1, rational=False, tolerance=None)`

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee $b \leq \text{Im } z \leq d$, $a \leq \text{Re } z \leq c$

When rational is True, a rational approximation to a random number is obtained within specified tolerance, if any.

`sympy.utilities.randtest.test_derivative_numerically(f, z, tol=1e-06, a=2, b=-1, c=3, d=1)`

Test numerically that the symbolically computed derivative of f with respect to z is correct.

This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> from sympy import sin  
>>> from sympy.abc import x  
>>> from sympy.utilities.randtest import test_derivative_numerically as td  
>>> td(sin(x), x)  
True
```

`sympy.utilities.randtest.verify_numerically(f, g, z=None, tol=1e-06, a=2, b=-1, c=3, d=1)`

Test numerically that f and g agree when evaluated in the argument z .

If z is None, all symbols will be tested. This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> from sympy import sin, cos  
>>> from sympy.abc import x  
>>> from sympy.utilities.randtest import verify_numerically as tn  
>>> tn(sin(x)**2 + cos(x)**2, 1, x)  
True
```

5.36.12 Run Tests

This is our testing framework.

Goals:

- it should be compatible with `py.test` and operate very similarly (or identically)
- doesn't require any external dependencies
- preferably all the functionality should be in this file only
- no magic, just import the test file and execute the test functions, that's it
- portable

class `sympy.utilities.runtests.PyTestReporter(verbose=False, tb='short', colors=True, force_colors=False, split=None)`

Py.test like reporter. Should produce output identical to `py.test`.

write(text, color="", align='left', width=None, force_colors=False)
Prints a text on the screen.

It uses `sys.stdout.write()`, so no readline library is necessary.

Parameters **color** : choose from the colors below, "" means default color

align : "left"/"right", "left" is a normal print, "right" is aligned on the right-hand side of the screen, filled with spaces if necessary

width : the screen width

class `sympy.utilities.runtests.Reporter`

Parent class for all reporters.

class `sympy.utilities.runtests.SymPyDocTestFinder(verbose=False, parser=<doctest.DocTestParser object>, recurse=True, exclude_empty=True)`

A class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. Doctests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

Modified from `doctest`'s version to look harder for code that appears comes from a different module. For example, the `@vectorize` decorator makes it look like functions come from `multidimensional.py` even though their code exists elsewhere.

class `sympy.utilities.runtests.SymPyDocTestRunner(checker=None, verbose=None, optionflags=0)`

A class used to run DocTest test cases, and accumulate statistics. The `run` method is used to process a single DocTest case. It returns a tuple `(f, t)`, where `t` is the number of test cases tried, and `f` is the number of test cases that failed.

Modified from the `doctest` version to not reset the `sys.displayhook` (see issue 5140).

See the docstring of the original `DocTestRunner` for more information.

run(test, compileflags=None, out=None, clear_globs=True)

Run the examples in `test`, and display the results using the `writer` function `out`.

The examples are run in the namespace `test.globs`. If `clear_globs` is true (the default), then this namespace will be cleared after the test runs, to help with garbage

collection. If you would like to examine the namespace after the test completes, then use `clear_globs=False`.

`compileflags` gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to globs.

The output of each example is checked using `SymPyDocTestRunner.check_output`, and the results are formatted by the `SymPyDocTestRunner.report_*` methods.

class `sympy.utilities.runtests.SymPyOutputChecker`

Compared to the `OutputChecker` from the `stdlib` our `OutputChecker` class supports numerical comparison of floats occurring in the output of the doctest examples

check_output(*want*, *got*, *optionflags*)

Return True iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See the documentation for *TestRunner* for more information about option flags.

`sympy.utilities.runtests.SymPyTestResults`

alias of `sympy.utilities.runtests.TestResults`

`sympy.utilities.runtests.convert_to_native_paths`(*lst*)

Converts a list of '/' separated paths into a list of native (`os.sep` separated) paths and converts to lowercase if the system is case insensitive.

`sympy.utilities.runtests.doctest`(**paths*, ***kwargs*)

Runs doctests in all *.py files in the sympy directory which match any of the given strings in *paths* or all tests if *paths*=[].

Notes:

- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.doctest()
```

Run one file:

```
>>> sympy.doctest("sympy/core/basic.py")
>>> sympy.doctest("polynomial.rst")
```

Run all tests in `sympy/functions/` and some particular file:

```
>>> sympy.doctest("/functions", "basic.py")
```

Run any file having polynomial in its name, `doc/src/modules/polynomial.rst`, `sympy/functions/special/polynomials.py`, and `sympy/polys/polynomial.py`:

```
>>> sympy.doctest("polynomial")
```

The `split` option can be passed to split the test run into parts. The `split` currently only splits the test files, though this may change in the future. `split` should be a string of the form 'a/b', which will run part a of b. Note that the regular doctests and the Sphinx doctests are split independently. For instance, to run the first half of the test suite:

```
>>> sympy.doctest(split='1/2')
```

The `subprocess` and `verbose` options are the same as with the function `test()`. See the docstring of that function for more information.

`sympy.utilities.runtests.get_sympy_dir()`

Returns the root sympy directory and set the global value indicating whether the system is case sensitive or not.

`sympy.utilities.runtests.run_all_tests(test_args=(), test_kwargs=None, doctest_args=(), doctest_kwargs=None, examples_args=(), examples_kwargs=None)`

Run all tests.

Right now, this runs the regular tests (`bin/test`), the doctests (`bin/doctest`), the examples (`examples/all.py`), and the sage tests (see `sympy/external/tests/test_sage.py`).

This is what `setup.py test` uses.

You can pass arguments and keyword arguments to the test functions that support them (for now, `test`, `doctest`, and the examples). See the docstrings of those functions for a description of the available options.

For example, to run the solvers tests with colors turned off:

```
>>> from sympy.utilities.runtests import run_all_tests
>>> run_all_tests(test_args=("solvers",),
... test_kwargs={"colors:False"})
```

`sympy.utilities.runtests.run_in_subprocess_with_hash_randomization(function, function_args=(), function_kwargs=None, command='/opt/conda/envs/release-docs/bin/python', module='sympy.utilities.runtests', force=False)`

Run a function in a Python subprocess with hash randomization enabled.

If hash randomization is not supported by the version of Python given, it returns `False`. Otherwise, it returns the exit value of the command. The function is passed to `sys.exit()`, so the return value of the function will be the return value.

The environment variable `PYTHONHASHSEED` is used to seed Python's hash randomization. If it is set, this function will return `False`, because starting a new subprocess is unnecessary in that case. If it is not set, one is set at random, and the tests are run. Note that if this environment variable is set when Python starts, hash randomization is automatically enabled. To force a subprocess to be created even if `PYTHONHASHSEED`

is set, pass `force=True`. This flag will not force a subprocess in Python versions that do not support hash randomization (see below), because those versions of Python do not support the `-R` flag.

function should be a string name of a function that is importable from the module `module`, like `"_test"`. The default for `module` is `"sympy.utilities.runtests"`. `function_args` and `function_kwargs` should be a repr-able tuple and dict, respectively. The default Python command is `sys.executable`, which is the currently running Python command.

This function is necessary because the seed for hash randomization must be set by the environment variable before Python starts. Hence, in order to use a predetermined seed for tests, we must start Python in a separate subprocess.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

Examples

```
>>> from sympy.utilities.runtests import (
...     run_in_subprocess_with_hash_randomization)
>>> # run the core tests in verbose mode
>>> run_in_subprocess_with_hash_randomization("_test",
...     function_args=("core",),
...     function_kwargs={'verbose': True})
# Will return 0 if sys.executable supports hash randomization and tests
# pass, 1 if they fail, and False if it does not support hash
# randomization.
```

`sympy.utilities.runtests.split_list(l, split, density=None)`

Splits a list into part a of b

`split` should be a string of the form `'a/b'`. For instance, `'1/3'` would give the split one of three.

If the length of the list is not divisible by the number of splits, the last split will have more items.

density may be specified as a list. If specified, tests will be balanced so that each split has as equal-as-possible amount of mass according to *density*.

```
>>> from sympy.utilities.runtests import split_list
>>> a = list(range(10))
>>> split_list(a, '1/3')
[0, 1, 2]
>>> split_list(a, '2/3')
[3, 4, 5]
>>> split_list(a, '3/3')
[6, 7, 8, 9]
```

`sympy.utilities.runtests.symptestfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extra_globs=None, raise_on_error=False, parser=<doctest.DocTestParser object>, encoding=None)`

Test examples in the given file. Return (`#failures`, `#tests`).

Optional keyword arg `module_relative` specifies how filenames should be interpreted:

- If `module_relative` is `True` (the default), then `filename` specifies a module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure os-independence, `filename` should use `"/"` characters to separate path segments, and should not be an absolute path (i.e., it may not begin with `"/"`).
- If `module_relative` is `False`, then `filename` specifies an os-specific path. The path may be absolute or relative (to the current working directory).

Optional keyword arg `name` gives the name of the test; by default use the file's basename.

Optional keyword argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module relative filename. If no `package` is specified, then the calling module's directory is used as the base directory for module relative filenames. It is an error to specify `package` if `module_relative` is `False`.

Optional keyword arg `globals` gives a dict to be used as the globals when executing examples; by default, use `{}`. A copy of this dict is actually used for each docstring, so that each docstring's examples start with a clean slate.

Optional keyword arg `extraglobs` gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used.

Optional keyword arg `verbose` prints lots of stuff if `true`, prints only failures if `false`; by default, it's `true` iff `"-v"` is in `sys.argv`.

Optional keyword arg `report` prints a summary at the end when `true`, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg `optionflags` or's together module constants, and defaults to 0. Possible values (see the docs for details):

- `DONT_ACCEPT_TRUE_FOR_1`
- `DONT_ACCEPT_BLANKLINE`
- `NORMALIZE_WHITESPACE`
- `ELLIPSIS`
- `SKIP`
- `IGNORE_EXCEPTION_DETAIL`
- `REPORT_UDIFF`
- `REPORT_CDIFF`
- `REPORT_NDIFF`
- `REPORT_ONLY_FIRST_FAILURE`

Optional keyword arg `raise_on_error` raises an exception on the first unexpected exception or failure. This allows failures to be post-mortem debugged.

Optional keyword arg `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files.

Optional keyword arg `encoding` specifies an encoding that should be used to convert the file to unicode.

Advanced tomfoolery: `testmod` runs methods of a local instance of class `doctest.Tester`, then merges the results into (or creates) global `Tester` instance `doctest.master`. Methods of `doctest.master` can be called directly too, if you want to do something unusual. Passing

report=0 to testmod is especially useful then, to delay displaying a summary. Invoke doctest.master.summarize(verbose) when you're done fiddling.

`sympy.utilities.runtests.test(*paths, **kwargs)`

Run tests in the specified test_*.py files.

Tests in a particular test_*.py file are run if any of the given strings in paths matches a part of the test file's path. If paths=[], tests in all test_*.py files are run.

Notes:

- If sort=False, tests are run in random order (not default).
- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

Explanation of test results

Out-put	Meaning
.	passed
F	failed
X	XPassed (expected to fail but passed)
f	XFAILed (expected to fail and indeed failed)
s	skipped
w	slow
T	timeout (e.g., when --timeout is used)
K	KeyboardInterrupt (when running the slow tests with --slow, you can interrupt one of them without killing the test runner)

Colors have no additional meaning and are used just to facilitate interpreting the output.

Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.test()
```

Run one file:

```
>>> sympy.test("sympy/core/tests/test_basic.py")
>>> sympy.test("_basic")
```

Run all tests in sympy/functions/ and some particular file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
...           "sympy/functions")
```

Run all tests in sympy/core and sympy/utilities:

```
>>> sympy.test("/core", "/util")
```

Run specific test from a file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
...           kw="test_equality")
```

Run specific test from any file:

```
>>> sympy.test(kw="subs")
```

Run the tests with verbose mode on:

```
>>> sympy.test(verbose=True)
```

Don't sort the test output:

```
>>> sympy.test(sort=False)
```

Turn on post-mortem pdb:

```
>>> sympy.test(pdb=True)
```

Turn off colors:

```
>>> sympy.test(colors=False)
```

Force colors, even when the output is not to a terminal (this is useful, e.g., if you are piping to `less -r` and you still want colors)

```
>>> sympy.test(force_colors=False)
```

The traceback verbosity can be set to "short" or "no" (default is "short")

```
>>> sympy.test(tb='no')
```

The `split` option can be passed to split the test run into parts. The `split` currently only splits the test files, though this may change in the future. `split` should be a string of the form 'a/b', which will run part a of b. For instance, to run the first half of the test suite:

```
>>> sympy.test(split='1/2')
```

The `time_balance` option can be passed in conjunction with `split`. If `time_balance=True` (the default for `sympy.test`), `sympy` will attempt to split the tests such that each split takes equal time. This heuristic for balancing is based on pre-recorded test data.

```
>>> sympy.test(split='1/2', time_balance=True)
```

You can disable running the tests in a separate subprocess using `subprocess=False`. This is done to support seeding hash randomization, which is enabled by default in the Python versions where it is supported. If `subprocess=False`, hash randomization is enabled/disabled according to whether it has been enabled or not in the calling Python process. However, even if it is enabled, the seed cannot be printed unless it is called from a new Python process.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

If hash randomization is not supported `subprocess=False` is used automatically.

```
>>> sympy.test(subprocess=False)
```

To set the hash randomization seed, set the environment variable PYTHONHASHSEED before running the tests. This can be done from within Python using

```
>>> import os
>>> os.environ['PYTHONHASHSEED'] = '42'
```

Or from the command line using

```
$ PYTHONHASHSEED=42 ./bin/test
```

If the seed is not set, a random seed will be chosen.

Note that to reproduce the same hash values, you must use both the same seed as well as the same architecture (32-bit vs. 64-bit).

5.36.13 Source Code Inspection

This module adds several functions for interactive source code inspection.

`sympy.utilities.source.get_class(lookup_view)`

Convert a string version of a class name to the object.

For example, `get_class('sympy.core.Basic')` will return class `Basic` located in module `sympy.core`

`sympy.utilities.source.get_mod_func(callback)`

splits the string path to a class into a string path to the module and the name of the class.

For example:

```
>>> from sympy.utilities.source import get_mod_func
>>> get_mod_func('sympy.core.basic.Basic')
('sympy.core.basic', 'Basic')
```

`sympy.utilities.source.source(object)`

Prints the source code of a given object.

5.36.14 Timing Utilities

Simple tools for timing functions' execution, when IPython is not available.

`sympy.utilities.timeutils.timed(func, setup='pass', limit=None)`

Adaptively measure execution time of a function.

5.37 Parsing input

5.37.1 Parsing Functions Reference

`sympy.parsing.sympy_parser.parse_expr(s, local_dict=None, transformations=(<function lambda_notation>, <function auto_symbol>, <function repeated_decimals>, <function auto_number>, <function factorial_notation>), global_dict=None, evaluate=True)`

Converts the string `s` to a SymPy expression, in `local_dict`

Parameters `s` : str

The string to parse.

local_dict : dict, optional

A dictionary of local variables to use when parsing.

global_dict : dict, optional

A dictionary of global variables. By default, this is initialized with from `sympy import *`; provide this parameter to override this behavior (for instance, to parse "Q & S").

transformations : tuple, optional

A tuple of transformation functions used to modify the tokens of the parsed expression before evaluation. The default transformations convert numeric literals into their SymPy equivalents, convert undefined variables into SymPy symbols, and allow the use of standard mathematical factorial notation (e.g. `x!`).

evaluate : bool, optional

When False, the order of the arguments will remain as they were in the string and automatic simplification that would normally occur is suppressed. (see examples)

See also:

[stringify_expr](#) (page 1456), [eval_expr](#) (page 1456), [standard_transformations](#) (page 1456), [implicit_multiplication_application](#) (page 1457)

Examples

```
>>> from sympy.parsing.sympy_parser import parse_expr
>>> parse_expr("1/2")
1/2
>>> type(_)
<class 'sympy.core.numbers.Half'>
>>> from sympy.parsing.sympy_parser import standard_transformations,\
... implicit_multiplication_application
>>> transformations = (standard_transformations +
... (implicit_multiplication_application,))
>>> parse_expr("2x", transformations=transformations)
2*x
```

When `evaluate=False`, some automatic simplifications will not occur:

```
>>> parse_expr("2**3"), parse_expr("2**3", evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows one to tell exactly how the expression was entered:

```
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

`sympy.parsing.sympy_parser.stringify_expr(s, local_dict, global_dict, transformations)`

Converts the string `s` to Python code, in `local_dict`

Generally, `parse_expr` should be used.

`sympy.parsing.sympy_parser.eval_expr(code, local_dict, global_dict)`

Evaluate Python code generated by `stringify_expr`.

Generally, `parse_expr` should be used.

`sympy.parsing.maxima.parse_maxima(str, globals=None, name_dict={})`

`sympy.parsing.mathematica.mathematica(s, additional_translations=None)`

Users can add their own translation dictionary # Example In [1]: `mathematica('Log3[9]', {'Log3[x]': 'log(x,3)'})` Out[1]: 2 In [2]: `mathematica('F[7,5,3]', {'F[x]': 'Max(*x)*Min(*x)'})` Out[2]: 21 variable-length argument needs " character

5.37.2 Parsing Transformations Reference

A transformation is a function that accepts the arguments `tokens`, `local_dict`, `global_dict` and returns a list of transformed tokens. They can be used by passing a list of functions to `parse_expr()` and are applied in the order given.

`sympy.parsing.sympy_parser.standard_transformations = (<function lambda_notation>, <function`

Standard transformations for `parse_expr()` (page 1455). Inserts calls to `Symbol`, `Integer`, and other SymPy datatypes and allows the use of standard factorial notation (e.g. `x!`).

`sympy.parsing.sympy_parser.split_symbols(tokens, local_dict, global_dict)`

Splits symbol names for implicit multiplication.

Intended to let expressions like `xyz` be parsed as `x*y*z`. Does not split Greek character names, so `theta` will not become `t*h*e*t*a`. Generally this should be used with `implicit_multiplication`.

`sympy.parsing.sympy_parser.split_symbols_custom(predicate)`

Creates a transformation that splits symbol names.

`predicate` should return `True` if the symbol name is to be split.

For instance, to retain the default behavior but avoid splitting certain symbol names, a predicate like this would work:

```
>>> from sympy.parsing.sympy_parser import (parse_expr, _tokenSplittable,
... standard_transformations, implicit_multiplication,
... split_symbols_custom)
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...         return _tokenSplittable(symbol)
...     return False
...
>>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations +
... (transformation, implicit_multiplication))
unsplittable
```

`sympy.parsing.sympy_parser.implicit_multiplication(result, local_dict, global_dict)`

Makes the multiplication operator optional in most cases.

Use this before `implicit_application()` (page 1457), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication)
>>> transformations = standard_transformations + (implicit_multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y
```

`sympy.parsing.sympy_parser.implicit_application(result, local_dict, global_dict)`

Makes parentheses optional in some cases for function calls.

Use this after `implicit_multiplication()` (page 1457), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_application)
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)
```

`sympy.parsing.sympy_parser.function_exponentiation(tokens, local_dict, global_dict)`

Allows functions to be exponentiated, e.g. `cos**2(x)`.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, function_exponentiation)
>>> transformations = standard_transformations + (function_exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

`sympy.parsing.sympy_parser.implicit_multiplication_application(result, local_dict, global_dict)`

Allows a slightly relaxed syntax.

- Parentheses for single-argument method calls are optional.
- Multiplication is implicit.
- Symbol names can be split (i.e. spaces are not needed between symbols).
- Functions can be exponentiated.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication_application)
>>> parse_expr("10sin**2 x**2 + 3xyz + tan theta",
... transformations=(standard_transformations +
... (implicit_multiplication_application,)))
3*x*y*z + 10*sin(x**2)**2 + tan(theta)
```

`sympy.parsing.sympy_parser.rationalize(tokens, local_dict, global_dict)`
Converts floats into Rational. Run AFTER `auto_number`.

`sympy.parsing.sympy_parser.convert_xor(tokens, local_dict, global_dict)`
Treats XOR, ^, as exponentiation, **.

These are included in `:data:sympy.parsing.sympy_parser.standard_transformations` and generally don't need to be manually added by the user.

`sympy.parsing.sympy_parser.lambda_notation(tokens, local_dict, global_dict)`
Substitutes "lambda" with its SymPy equivalent `Lambda()`. However, the conversion doesn't take place if only "lambda" is passed because that is a syntax error.

`sympy.parsing.sympy_parser.auto_symbol(tokens, local_dict, global_dict)`
Inserts calls to `Symbol` for undefined variables.

`sympy.parsing.sympy_parser.repeated_decimals(tokens, local_dict, global_dict)`
Allows `0.2[1]` notation to represent the repeated decimal `0.2111...` (`19/90`)
Run this before `auto_number`.

`sympy.parsing.sympy_parser.auto_number(tokens, local_dict, global_dict)`
Converts numeric literals to use SymPy equivalents.
Complex numbers use `I`, integer literals use `Integer`, and float literals use `Float`.

`sympy.parsing.sympy_parser.factorial_notation(tokens, local_dict, global_dict)`
Allows standard notation for factorial.

5.37.3 Experimental $\LaTeX$ Parsing

$\LaTeX$ parsing was ported from `latex2sympy`. While functional and its API should remain stable, the parsing behavior or backend may change in future releases.

5.37.4 $\text{\LaTeX}$ Parsing Caveats

The current implementation is experimental. The behavior, parser backend and API might change in the future. Unlike some of the other parsers, $\text{\LaTeX}$ is designed as a type-setting language, not a computer algebra system and so can contain typographical conventions that might be interpreted multiple ways.

In its current definition, the parser will at times will fail to fully parse the expression, but not throw a warning:

```
parse_latex(r'x -')
```

Will simply find x . What is covered by this behavior will almost certainly change between releases, and become stricter, more relaxed, or some mix.

5.37.5 $\text{\LaTeX}$ Parsing Functions Reference

`sympy.parsing.latex.parse_latex(s)`
Converts the string s to a SymPy Expr

Parameters s : str

The $\text{\LaTeX}$ string to parse. In Python source containing $\text{\LaTeX}$, **raw strings**

(denoted with `'r''`, like this one) are preferred, as $\text{\LaTeX}$ makes liberal

use of the `'''` character, which would trigger escaping in normal Python

strings.

```
>>> from sympy.parsing.latex import parse_latex # doctest: +SKIP
```

```
>>> expr = parse_latex(r"frac {1 + sqrt {a}} {b}") # doctest: +SKIP
```

```
>>> expr # doctest: +SKIP
```

```
(sqrt(a) + 1)/b
```

```
>>> expr.evalf(4, subs=dict(a=5, b=2)) # doctest: +SKIP
```

```
1.618
```

5.37.6 $\text{\LaTeX}$ Parsing Exceptions Reference

`class sympy.parsing.latex.LaTeXParsingError`

5.37.7 Runtime Installation

The currently-packaged parser backend is partially generated with [ANTLR4](#), but to use the parser, you only need the `antlr4` Python package available.

Depending on your package manager, you can install the right package with, for example, `pip3` (Python 3 only):

```
$ pip3 install antlr4-python3-runtime
```

or pip (Python 2 only):

```
$ pip install antlr4-python2-runtime
```

or conda (Python 2 or Python 3):

```
$ conda install --channel=conda-forge antlr-python-runtime
```

5.38 Calculus

Calculus-related methods.

This module implements a method to find Euler-Lagrange Equations for given Lagrangian.

`sympy.calculus.euler.euler_equations(L, funcs=(), vars=())`

Find the Euler-Lagrange equations [R22] (page 1872) for a given Lagrangian.

Parameters **L** : Expr

The Lagrangian that should be a function of the functions listed in the second argument and their derivatives.

For example, in the case of two functions $f(x, y)$, $g(x, y)$ and two independent variables x , y the Lagrangian would have the form:

$$L\left(f(x, y), g(x, y), \frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y}, \frac{\partial g(x, y)}{\partial x}, \frac{\partial g(x, y)}{\partial y}, x, y\right)$$

In many cases it is not necessary to provide anything, except the Lagrangian, it will be auto-detected (and an error raised if this couldn't be done).

funcs : Function or an iterable of Functions

The functions that the Lagrangian depends on. The Euler equations are differential equations for each of these functions.

vars : Symbol or an iterable of Symbols

The Symbols that are the independent variables of the functions.

Returns **eqns** : list of Eq

The list of differential equations, one for each function.

References

[R22] (page 1872)

Examples

```

>>> from sympy import Symbol, Function
>>> from sympy.calculus.euler import euler_equations
>>> x = Function('x')
>>> t = Symbol('t')
>>> L = (x(t).diff(t))**2/2 - x(t)**2/2
>>> euler_equations(L, x(t), t)
[Eq(-x(t) - Derivative(x(t), (t, 2)), 0)]
>>> u = Function('u')
>>> x = Symbol('x')
>>> L = (u(t, x).diff(t))**2/2 - (u(t, x).diff(x))**2/2
>>> euler_equations(L, u(t, x), [t, x])
[Eq(-Derivative(u(t, x), (t, 2)) + Derivative(u(t, x), (x, 2)), 0)]

```

5.38.1 Singularities

This module implements algorithms for finding singularities for a function and identifying types of functions.

The differential calculus methods in this module include methods to identify the following function types in the given Interval: - Increasing - Strictly Increasing - Decreasing - Strictly Decreasing - Monotonic

`sympy.calculus.singularities.is_decreasing(expression, interval=Reals, symbol=None)`

Return whether the function is decreasing in the given interval.

Examples

```

>>> from sympy import is_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_decreasing(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_decreasing(-x**2 + y, Interval(-oo, 0), x)
False

```

`sympy.calculus.singularities.is_increasing(expression, interval=Reals, symbol=None)`

Return whether the function is increasing in the given interval.

Examples

```

>>> from sympy import is_increasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_increasing(x**3 - 3*x**2 + 4*x, S.Reals)

```

(continues on next page)

(continued from previous page)

```

True
>>> is_increasing(-x**2, Interval(-oo, 0))
True
>>> is_increasing(-x**2, Interval(0, oo))
False
>>> is_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval(-2, 3))
False
>>> is_increasing(x**2 + y, Interval(1, 2), x)
True

```

`sympy.calculus.singularities.is_monotonic`(expression, interval=Reals, sym-
bol=None)
Return whether the function is monotonic in the given interval.

Examples

```

>>> from sympy import is_monotonic
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_monotonic(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_monotonic(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_monotonic(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_monotonic(-x**2, S.Reals)
False
>>> is_monotonic(x**2 + y + 1, Interval(1, 2), x)
True

```

`sympy.calculus.singularities.is_strictly_decreasing`(expression, interval=Reals, sym-
bol=None)
Return whether the function is strictly decreasing in the given interval.

Examples

```

>>> from sympy import is_strictly_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_strictly_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_strictly_decreasing(-x**2 + y, Interval(-oo, 0), x)
False

```

`sympy.calculus.singularities.is_strictly_increasing`(expression, interval=Reals, sym-
bol=None)
Return whether the function is strictly increasing in the given interval.

Examples

```
>>> from sympy import is_strictly_increasing
>>> from sympy.abc import x, y
>>> from sympy import Interval, oo
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Ropen(-oo, -2))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Lopen(3, oo))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.open(-2, 3))
False
>>> is_strictly_increasing(-x**2, Interval(0, oo))
False
>>> is_strictly_increasing(-x**2 + y, Interval(-oo, 0), x)
False
```

`sympy.calculus.singularities.monotonicity_helper(expression, predicate, interval=Reals, symbol=None)`

Helper function for functions checking function monotonicity.

It returns a boolean indicating whether the interval in which the function's derivative satisfies given predicate is a superset of the given interval.

`sympy.calculus.singularities.singularities(expression, symbol)`

Find singularities of a given function.

Currently supported functions are: - univariate rational (real or complex) functions

Notes

This function does not find nonisolated singularities nor does it find branch points of the expression.

References

[R23] (page 1872)

Examples

```
>>> from sympy.calculus.singularities import singularities
>>> from sympy import Symbol
>>> x = Symbol('x', real=True)
>>> y = Symbol('y', real=False)
>>> singularities(x**2 + x + 1, x)
EmptySet()
>>> singularities(1/(x + 1), x)
{-1}
>>> singularities(1/(y**2 + 1), y)
{-I, I}
>>> singularities(1/(y**3 + 1), y)
{-1, 1/2 - sqrt(3)*I/2, 1/2 + sqrt(3)*I/2}
```

5.38.2 Finite difference weights

This module implements an algorithm for efficient generation of finite difference weights for ordinary differentials of functions for derivatives from 0 (interpolation) up to arbitrary order.

The core algorithm is provided in the finite difference weight generating function (`finite_diff_weights`), and two convenience functions are provided for:

- **estimating a derivative (or interpolate) directly from a series of points** is also provided (`apply_finite_diff`).
- **differentiating by using finite difference approximations** (`differentiate_finite`).

`sympy.calculus.finite_diff.apply_finite_diff(order, x_list, y_list, x0=0)`

Calculates the finite difference approximation of the derivative of requested order at x_0 from points provided in `x_list` and `y_list`.

Parameters `order`: **int**

order of derivative to approximate. 0 corresponds to interpolation.

x_list: **sequence**

Sequence of (unique) values for the independent variable.

y_list: **sequence**

The function value at corresponding values for the independent variable in `x_list`.

x0: **Number or Symbol**

At what value of the independent variable the derivative should be evaluated. Defaults to $S(0)$.

Returns `sympy.core.add.Add` or `sympy.core.numbers.Number`

The finite difference expression approximating the requested derivative order at x_0 .

See also:

`sympy.calculus.finite_diff.finite_diff_weights` (page 1467)

Notes

Order = 0 corresponds to interpolation. Only supply so many points you think makes sense to around x_0 when extracting the derivative (the function need to be well behaved within that region). Also beware of Runge's phenomenon.

References

Fortran 90 implementation with Python interface for numerics: [fnitediff](#)

Examples

```
>>> from sympy.calculus import apply_finite_diff
>>> cube = lambda arg: (1.0*arg)**3
>>> xlist = range(-3,3+1)
>>> apply_finite_diff(2, xlist, map(cube, xlist), 2) - 12
-3.55271367880050e-15
```

we see that the example above only contain rounding errors. `apply_finite_diff` can also be used on more abstract objects:

```
>>> from sympy import IndexedBase, Idx
>>> from sympy.calculus import apply_finite_diff
>>> x, y = map(IndexedBase, 'xy')
>>> i = Idx('i')
>>> x_list, y_list = zip(*[(x[i+j], y[i+j]) for j in range(-1,2)])
>>> apply_finite_diff(1, x_list, y_list, x[i])
((x[i + 1] - x[i])/(-x[i - 1] + x[i]) - 1)*y[i]/(x[i + 1] - x[i]) - (x[i + 1] -
↪ x[i])*y[i - 1]/((x[i + 1] - x[i - 1])*(-x[i - 1] + x[i])) + (-x[i - 1] +
↪ x[i])*y[i + 1]/((x[i + 1] - x[i - 1])*(x[i + 1] - x[i]))
```

`sympy.calculus.finite_diff.as_finite_diff(derivative, points=1, x0=None, wrt=None)`

Returns an approximation of a derivative of a function in the form of a finite difference formula. The expression is a weighted sum of the function at a number of discrete values of (one of) the independent variable(s).

Parameters derivative: a Derivative instance

points: sequence or coefficient, optional

If sequence: discrete values (length $\geq$ order+1) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length order+1 centered around `x0`. default: 1 (step-size 1)

x0: number or Symbol, optional

the value of the independent variable (`wrt`) at which the derivative is to be approximated. Default: same as `wrt`.

wrt: Symbol, optional

“with respect to” the variable for which the (partial) derivative is to be approximated for. If not provided it is required that the Derivative is ordinary. Default: None.

See also:

`sympy.calculus.finite_diff.apply_finite_diff` (page 1464), `sympy.calculus.finite_diff.finite_diff_weights` (page 1467)

Examples

```
>>> from sympy import symbols, Function, exp, sqrt, Symbol, as_finite_diff
>>> from sympy.utilities.exceptions import SymPyDeprecationWarning
>>> import warnings
>>> warnings.simplefilter("ignore", SymPyDeprecationWarning)
>>> x, h = symbols('x h')
```

(continues on next page)

(continued from previous page)

```
>>> f = Function('f')
>>> as_finite_diff(f(x).diff(x))
-f(x - 1/2) + f(x + 1/2)
```

The default step size and number of points are 1 and order + 1 respectively. We can change the step size by passing a symbol as a parameter:

```
>>> as_finite_diff(f(x).diff(x), h)
-f(-h/2 + x)/h + f(h/2 + x)/h
```

We can also specify the discretized values to be used in a sequence:

```
>>> as_finite_diff(f(x).diff(x), [x, x+h, x+2*h])
-3*f(x)/(2*h) + 2*f(h + x)/h - f(2*h + x)/(2*h)
```

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around x_0 , but we can get an expression estimating the derivative at an offset:

```
>>> e, sq2 = exp(1), sqrt(2)
>>> xl = [x-h, x+h, x+e*h]
>>> as_finite_diff(f(x).diff(x, 1), xl, x+h*sq2)
2*h*((h + sqrt(2)*h)/(2*h) - (-sqrt(2)*h + h)/(2*h))*f(E*h + x)/((-h + E*h)*(h +
↪ E*h)) + (-(-sqrt(2)*h + h)/(2*h) - (-sqrt(2)*h + E*h)/(2*h))*f(-h + x)/(h +
↪ E*h) + (-(-h + sqrt(2)*h)/(2*h) + (-sqrt(2)*h + E*h)/(2*h))*f(h + x)/(-h + E*h)
```

Partial derivatives are also supported:

```
>>> y = Symbol('y')
>>> d2fdxdy=f(x,y).diff(x,y)
>>> as_finite_diff(d2fdxdy, wrt=x)
-Derivative(f(x - 1/2, y), y) + Derivative(f(x + 1/2, y), y)
```

`sympy.calculus.finite_diff.differentiate_finite(expr, *symbols, **kwargs)`

Differentiate `expr` and replace Derivatives with finite differences.

Parameters `expr` : expression

***symbols** : differentiate with respect to symbols

points: sequence or coefficient, optional

see `Derivative.as_finite_difference`

x0: number or Symbol, optional

see `Derivative.as_finite_difference`

wrt: Symbol, optional

see `Derivative.as_finite_difference`

evaluate : bool

kwarg passed on to `diff`, whether or not to evaluate the `Derivative` intermediately (default: `False`).

Examples

```
>>> from sympy import cos, sin, Function, differentiate_finite
>>> from sympy.abc import x, y, h
>>> f, g = Function('f'), Function('g')
>>> differentiate_finite(f(x)*g(x), x, points=[x-h, x+h])
-f(-h + x)*g(-h + x)/(2*h) + f(h + x)*g(h + x)/(2*h)
```

Note that the above form preserves the product rule in discrete form. If we want we can pass `evaluate=True` to get another form (which is usually not what we want):

```
>>> differentiate_finite(f(x)*g(x), x, points=[x-h, x+h], evaluate=True).
↪ simplify()
-((f(-h + x) - f(h + x))*g(x) + (g(-h + x) - g(h + x))*f(x))/(2*h)
```

`differentiate_finite` works on any expression:

```
>>> differentiate_finite(f(x) + sin(x), x, 2)
-2*f(x) + f(x - 1) + f(x + 1) - 2*sin(x) + sin(x - 1) + sin(x + 1)
>>> differentiate_finite(f(x) + sin(x), x, 2, evaluate=True)
-2*f(x) + f(x - 1) + f(x + 1) - sin(x)
>>> differentiate_finite(f(x, y), x, y)
f(x - 1/2, y - 1/2) - f(x - 1/2, y + 1/2) - f(x + 1/2, y - 1/2) + f(x + 1/2, y +
↪ 1/2)
```

`sympy.calculus.finite_diff.finite_diff_weights(order, x_list, x0=1)`

Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at `x0` of order 0, 1, ..., up to order using a recursive formula. Order of accuracy is at least `len(x_list) - order`, if `x_list` is defined correctly.

Parameters `order`: int

Up to what derivative order weights should be calculated. 0 corresponds to interpolation.

`x_list`: sequence

Sequence of (unique) values for the independent variable. It is useful (but not necessary) to order `x_list` from nearest to furthest from `x0`; see examples below.

`x0`: Number or Symbol

Root or value of the independent variable for which the finite difference weights should be generated. Default is `S.One`.

Returns list

A list of sublists, each corresponding to coefficients for increasing derivative order, and each containing lists of coefficients for increasing subsets of `x_list`.

See also:

[sympy.calculus.finite_diff.apply_finite_diff](#) (page 1464)

Notes

If weights for a finite difference approximation of 3rd order derivative is wanted, weights for 0th, 1st and 2nd order are calculated “for free”, so are formulae using subsets of

`x_list`. This is something one can take advantage of to save computational cost. Be aware that one should define `x_list` from nearest to farthest from `x0`. If not, subsets of `x_list` will yield poorer approximations, which might not grant an order of accuracy of `len(x_list) - order`.

References

[R24] (page 1872)

Examples

```
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> res = finite_diff_weights(1, [-S(1)/2, S(1)/2, S(3)/2, S(5)/2], 0)
>>> res
[[[1, 0, 0, 0],
  [1/2, 1/2, 0, 0],
  [3/8, 3/4, -1/8, 0],
  [5/16, 15/16, -5/16, 1/16]],
 [[0, 0, 0, 0],
  [-1, 1, 0, 0],
  [-1, 1, 0, 0],
  [-23/24, 7/8, 1/8, -1/24]]]
>>> res[0][-1] # FD weights for 0th derivative, using full x_list
[5/16, 15/16, -5/16, 1/16]
>>> res[1][-1] # FD weights for 1st derivative
[-23/24, 7/8, 1/8, -1/24]
>>> res[1][-2] # FD weights for 1st derivative, using x_list[:-1]
[-1, 1, 0, 0]
>>> res[1][-1][0] # FD weight for 1st deriv. for x_list[0]
-23/24
>>> res[1][-1][1] # FD weight for 1st deriv. for x_list[1], etc.
7/8
```

Each sublist contains the most accurate formula at the end. Note, that in the above example `res[1][1]` is the same as `res[1][2]`. Since `res[1][2]` has an order of accuracy of `len(x_list[:3]) - order = 3 - 1 = 2`, the same is true for `res[1][1]`!

```
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> res = finite_diff_weights(1, [S(0), S(1), -S(1), S(2), -S(2)], 0)[1]
>>> res
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [0, 1/2, -1/2, 0, 0],
 [-1/2, 1, -1/3, -1/6, 0],
 [0, 2/3, -2/3, -1/12, 1/12]]
>>> res[0] # no approximation possible, using x_list[0] only
[0, 0, 0, 0, 0]
>>> res[1] # classic forward step approximation
[-1, 1, 0, 0, 0]
>>> res[2] # classic centered approximation
[0, 1/2, -1/2, 0, 0]
>>> res[3:] # higher order approximations
[[-1/2, 1, -1/3, -1/6, 0], [0, 2/3, -2/3, -1/12, 1/12]]
```

Let us compare this to a differently defined `x_list`. Pay attention to `foo[i][k]` corresponding to the gridpoint defined by `x_list[k]`.

```
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> foo = finite_diff_weights(1, [-S(2), -S(1), S(0), S(1), S(2)], 0)[1]
>>> foo
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [1/2, -2, 3/2, 0, 0],
 [1/6, -1, 1/2, 1/3, 0],
 [1/12, -2/3, 0, 2/3, -1/12]]
>>> foo[1] # not the same and of lower accuracy as res[1]!
[-1, 1, 0, 0, 0]
>>> foo[2] # classic double backward step approximation
[1/2, -2, 3/2, 0, 0]
>>> foo[4] # the same as res[4]
[1/12, -2/3, 0, 2/3, -1/12]
```

Note that, unless you plan on using approximations based on subsets of `x_list`, the order of gridpoints does not matter.

The capability to generate weights at arbitrary points can be used e.g. to minimize Runge's phenomenon by using Chebyshev nodes:

```
>>> from sympy import cos, symbols, pi, simplify
>>> from sympy.calculus import finite_diff_weights
>>> N, (h, x) = 4, symbols('h x')
>>> x_list = [x+h*cos(i*pi/(N)) for i in range(N,-1,-1)] # chebyshev nodes
>>> print(x_list)
[-h + x, -sqrt(2)*h/2 + x, x, sqrt(2)*h/2 + x, h + x]
>>> mycoeffs = finite_diff_weights(1, x_list, 0)[1][4]
>>> [simplify(c) for c in mycoeffs]
[(h**3/2 + h**2*x - 3*h*x**2 - 4*x**3)/h**4,
 (-sqrt(2)*h**3 - 4*h**2*x + 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 6*x/h**2 - 8*x**3/h**4,
 (sqrt(2)*h**3 - 4*h**2*x - 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 (-h**3/2 + h**2*x + 3*h*x**2 - 4*x**3)/h**4]
```

5.39 Physics Module

A module that helps solving problems in physics

5.39.1 Contents

Hydrogen Wavefunctions

`sympy.physics.hydrogen.E_nl(n, Z=1)`

Returns the energy of the state (n, l) in Hartree atomic units.

The energy doesn't depend on "l".

Examples

```
>>> from sympy import var
>>> from sympy.physics.hydrogen import E_nl
>>> var("n Z")
(n, Z)
>>> E_nl(n, Z)
-Z**2/(2*n**2)
>>> E_nl(1)
-1/2
>>> E_nl(2)
-1/8
>>> E_nl(3)
-1/18
>>> E_nl(3, 47)
-2209/18
```

`sympy.physics.hydrogen.E_nl_dirac(n, l, spin_up=True, Z=1, c=137.035999037000)`

Returns the relativistic energy of the state (n, l, spin) in Hartree atomic units.

The energy is calculated from the Dirac equation. The rest mass energy is not included.

n, l quantum numbers 'n' and 'l'

spin_up True if the electron spin is up (default), otherwise down

Z atomic number (1 for Hydrogen, 2 for Helium, ...)

c speed of light in atomic units. Default value is 137.035999037, taken from: <http://arxiv.org/abs/1012.3627>

Examples

```
>>> from sympy.physics.hydrogen import E_nl_dirac
>>> E_nl_dirac(1, 0)
-0.500006656595360
```

```
>>> E_nl_dirac(2, 0)
-0.125002080189006
>>> E_nl_dirac(2, 1)
-0.125000416028342
>>> E_nl_dirac(2, 1, False)
-0.125002080189006
```

```
>>> E_nl_dirac(3, 0)
-0.0555562951740285
>>> E_nl_dirac(3, 1)
-0.0555558020932949
>>> E_nl_dirac(3, 1, False)
-0.0555562951740285
>>> E_nl_dirac(3, 2)
-0.0555556377366884
>>> E_nl_dirac(3, 2, False)
-0.0555558020932949
```

`sympy.physics.hydrogen.Psi_nlm(n, l, m, r, phi, theta, Z=1)`

Returns the Hydrogen wave function ψ_{nlm} . It's the product of the radial wavefunction R_{nl} and the spherical harmonic Y_l^m .

n, l, m quantum numbers 'n', 'l' and 'm'

r radial coordinate

phi azimuthal angle

theta polar angle

Z atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.

Examples

```
>>> from sympy.physics.hydrogen import Psi_nlm
>>> from sympy import Symbol
>>> r=Symbol("r", real=True, positive=True)
>>> phi=Symbol("phi", real=True)
>>> theta=Symbol("theta", real=True)
>>> Z=Symbol("Z", positive=True, integer=True, nonzero=True)
>>> Psi_nlm(1,0,0,r,phi,theta,Z)
Z**(3/2)*exp(-Z*r)/sqrt(pi)
>>> Psi_nlm(2,1,1,r,phi,theta,Z)
-Z**(5/2)*r*exp(I*phi)*exp(-Z*r/2)*sin(theta)/(8*sqrt(pi))
```

Integrating the absolute square of a hydrogen wavefunction ψ_{nlm} over the whole space leads 1.

The normalization of the hydrogen wavefunctions Ψ_{nlm} is:

```
>>> from sympy import integrate, conjugate, pi, oo, sin
>>> wf=Psi_nlm(2,1,1,r,phi,theta,Z)
>>> abs_sqrd=wf*conjugate(wf)
>>> jacobi=r**2*sin(theta)
>>> integrate(abs_sqrd*jacobi, (r,0,oo), (phi,0,2*pi), (theta,0,pi))
1
```

`sympy.physics.hydrogen.R_nl(n, l, r, Z=1)`

Returns the Hydrogen radial wavefunction R_{nl} .

n, l quantum numbers 'n' and 'l'

r radial coordinate

Z atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.

Examples

```
>>> from sympy.physics.hydrogen import R_nl
>>> from sympy import var
>>> var("r Z")
(r, Z)
>>> R_nl(1, 0, r, Z)
```

(continues on next page)

(continued from previous page)

```

2*sqrt(Z**3)*exp(-Z*r)
>>> R_nl(2, 0, r, Z)
sqrt(2)*(-Z*r + 2)*sqrt(Z**3)*exp(-Z*r/2)/4
>>> R_nl(2, 1, r, Z)
sqrt(6)*Z*r*sqrt(Z**3)*exp(-Z*r/2)/12

```

For Hydrogen atom, you can just use the default value of $Z=1$:

```

>>> R_nl(1, 0, r)
2*exp(-r)
>>> R_nl(2, 0, r)
sqrt(2)*(-r + 2)*exp(-r/2)/4
>>> R_nl(3, 0, r)
2*sqrt(3)*(2*r**2/9 - 2*r + 3)*exp(-r/3)/27

```

For Silver atom, you would use $Z=47$:

```

>>> R_nl(1, 0, r, Z=47)
94*sqrt(47)*exp(-47*r)
>>> R_nl(2, 0, r, Z=47)
47*sqrt(94)*(-47*r + 2)*exp(-47*r/2)/4
>>> R_nl(3, 0, r, Z=47)
94*sqrt(141)*(4418*r**2/9 - 94*r + 3)*exp(-47*r/3)/27

```

The normalization of the radial wavefunction is:

```

>>> from sympy import integrate, oo
>>> integrate(R_nl(1, 0, r)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 0, r)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 1, r)**2 * r**2, (r, 0, oo))
1

```

It holds for any atomic number:

```

>>> integrate(R_nl(1, 0, r, Z=2)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 0, r, Z=3)**2 * r**2, (r, 0, oo))
1
>>> integrate(R_nl(2, 1, r, Z=4)**2 * r**2, (r, 0, oo))
1

```

Matrices

Known matrices related to physics

`sympy.physics.matrices.mdft(n)`

Returns an expression of a discrete Fourier transform as a matrix multiplication. It is an $n \times n$ matrix.

References

[R422] (page 1872)

Examples

```
>>> from sympy.physics.matrices import mdft
>>> mdft(3)
Matrix([
[sqrt(3)/3,          sqrt(3)/3,          sqrt(3)/3],
[sqrt(3)/3, sqrt(3)*exp(-2*I*pi/3)/3, sqrt(3)*exp(-4*I*pi/3)/3],
[sqrt(3)/3, sqrt(3)*exp(-4*I*pi/3)/3, sqrt(3)*exp(-8*I*pi/3)/3]])
```

`sympy.physics.matrices.mgamma(mu, lower=False)`

Returns a Dirac gamma matrix γ^μ in the standard (Dirac) representation.

If you want γ_μ , use `gamma(mu, True)`.

We use a convention:

$$\gamma^5 = i \cdot \gamma^0 \cdot \gamma^1 \cdot \gamma^2 \cdot \gamma^3$$

$$\gamma_5 = i \cdot \gamma_0 \cdot \gamma_1 \cdot \gamma_2 \cdot \gamma_3 = -\gamma^5$$

References

[R423] (page 1872)

Examples

```
>>> from sympy.physics.matrices import mgamma
>>> mgamma(1)
Matrix([
[ 0,  0, 0, 1],
[ 0,  0, 1, 0],
[ 0, -1, 0, 0],
[-1,  0, 0, 0]])
```

`sympy.physics.matrices.msigma(i)`

Returns a Pauli matrix σ_i with $i = 1, 2, 3$

References

[R424] (page 1872)

Examples

```
>>> from sympy.physics.matrices import msigma
>>> msigma(1)
Matrix([
[0, 1],
[1, 0]])
```

`sympy.physics.matrices.pat_matrix(m, dx, dy, dz)`

Returns the Parallel Axis Theorem matrix to translate the inertia matrix a distance of (dx, dy, dz) for a body of mass m .

Examples

To translate a body having a mass of 2 units a distance of 1 unit along the x -axis we get:

```
>>> from sympy.physics.matrices import pat_matrix
>>> pat_matrix(2, 1, 0, 0)
Matrix([
[0, 0, 0],
[0, 2, 0],
[0, 0, 2]])
```

Pauli Algebra

This module implements Pauli algebra by subclassing Symbol. Only algebraic properties of Pauli matrices are used (we don't use the Matrix class).

See the documentation to the class Pauli for examples.

References

`sympy.physics.paulialgebra.evaluate_pauli_product(arg)`

Help function to evaluate Pauli matrices product with symbolic objects

Parameters **arg:** symbolic expression that contains Paulimatrices

Examples

```
>>> from sympy.physics.paulialgebra import Pauli, evaluate_pauli_product
>>> from sympy import I
>>> evaluate_pauli_product(I*Pauli(1)*Pauli(2))
-sigma3
```

```
>>> from sympy.abc import x,y
>>> evaluate_pauli_product(x**2*Pauli(2)*Pauli(1))
-I*x**2*sigma3
```

Quantum Harmonic Oscillator in 1-D

`sympy.physics.qho_1d.E_n(n, omega)`

Returns the Energy of the One-dimensional harmonic oscillator

n the “nodal” quantum number

omega the harmonic oscillator angular frequency

The unit of the returned value matches the unit of $\hbar\omega$, since the energy is calculated as:

$$E_n = \hbar\omega(n + 1/2)$$

Examples

```
>>> from sympy.physics.qho_1d import E_n
>>> from sympy import var
>>> var("x omega")
(x, omega)
>>> E_n(x, omega)
hbar*omega*(x + 1/2)
```

`sympy.physics.qho_1d.coherent_state(n, alpha)`

Returns $\langle n | \alpha \rangle$ for the coherent states of 1D harmonic oscillator. See http://en.wikipedia.org/wiki/Coherent_states

n the “nodal” quantum number

alpha the eigen value of annihilation operator

`sympy.physics.qho_1d.psi_n(n, x, m, omega)`

Returns the wavefunction ψ_n for the One-dimensional harmonic oscillator.

n the “nodal” quantum number. Corresponds to the number of nodes in the wavefunction. $n \geq 0$

x x coordinate

m mass of the particle

omega angular frequency of the oscillator

Examples

```
>>> from sympy.physics.qho_1d import psi_n
>>> from sympy import var
>>> var("x m omega")
(x, m, omega)
>>> psi_n(0, x, m, omega)
(m*omega)**(1/4)*exp(-m*omega*x**2/(2*hbar))/(hbar**(1/4)*pi**(1/4))
```

Quantum Harmonic Oscillator in 3-D

`sympy.physics.sho.E_nl(n, l, hw)`

Returns the Energy of an isotropic harmonic oscillator

n the “nodal” quantum number

l the orbital angular momentum

hw the harmonic oscillator parameter.

The unit of the returned value matches the unit of hw, since the energy is calculated as:

$$E_{nl} = (2*n + l + 3/2)*hw$$

Examples

```
>>> from sympy.physics.sho import E_nl
>>> from sympy import symbols
>>> x, y, z = symbols('x, y, z')
>>> E_nl(x, y, z)
z*(2*x + y + 3/2)
```

`sympy.physics.sho.R_nl(n, l, nu, r)`

Returns the radial wavefunction $R_{\{nl\}}$ for a 3d isotropic harmonic oscillator.

n the “nodal” quantum number. Corresponds to the number of nodes in the wavefunction. $n \geq 0$

l the quantum number for orbital angular momentum

nu mass-scaled frequency: $\nu = m\omega/(2\hbar)$ where m is the mass and ω the frequency of the oscillator. (in atomic units $\nu == \omega/2$)

r Radial coordinate

Examples

```
>>> from sympy.physics.sho import R_nl
>>> from sympy import var
>>> var("r nu l")
(r, nu, l)
>>> R_nl(0, 0, 1, r)
2*2**(3/4)*exp(-r**2)/pi**(1/4)
>>> R_nl(1, 0, 1, r)
4*2**(1/4)*sqrt(3)*(-2*r**2 + 3/2)*exp(-r**2)/(3*pi**(1/4))
```

l , ν and r may be symbolic:

```
>>> R_nl(0, 0, nu, r)
2*2**(3/4)*sqrt(nu**(3/2))*exp(-nu*r**2)/pi**(1/4)
>>> R_nl(0, l, 1, r)
r**l*sqrt(2**(l + 3/2)*2**(l + 2)/factorial2(2*l + 1))*exp(-r**2)/pi**(1/4)
```

The normalization of the radial wavefunction is:

```
>>> from sympy import Integral, oo
>>> Integral(R_nl(0, 0, 1, r)**2 * r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 0, 1, r)**2 * r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 1, 1, r)**2 * r**2, (r, 0, oo)).n()
1.00000000000000
```

Second Quantization

Second quantization operators and states for bosons.

This follows the formulation of Fetter and Welecka, “Quantum Theory of Many-Particle Systems.”

class `sympy.physics.secondquant.Dagger`

Hermitian conjugate of creation/annihilation operators.

Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

classmethod eval(arg)
Evaluates the Dagger instance.

Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

The eval() method is called automatically.

class sympy.physics.secondquant.**KroneckerDelta**

The discrete, or Kronecker, delta function.

A function that takes in two integers i and j . It returns 0 if i and j are not equal or it returns 1 if i and j are equal.

Parameters **i** : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

See also:

[eval](#) (page 1478), [sympy.functions.special.delta_functions.DiracDelta](#) (page 475)

References

[R447] (page 1872)

Examples

A simple example with integer indices:

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

classmethod `eval(i, j)`

Evaluates the discrete delta function.

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

indirect doctest

indices_contain_equal_information

Returns True if indices are either both above or below fermi.

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

is_above_fermi

True if Delta can be non-zero above fermi

See also:

[is_below_fermi](#) (page 1479), [is_only_below_fermi](#) (page 1480),
[is_only_above_fermi](#) (page 1479)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

is_below_fermi

True if Delta can be non-zero below fermi

See also:

[is_above_fermi](#) (page 1478), [is_only_above_fermi](#) (page 1479),
[is_only_below_fermi](#) (page 1480)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

is_only_above_fermi

True if Delta is restricted to above fermi

See also:

[is_above_fermi](#) (page 1478), [is_below_fermi](#) (page 1479), [is_only_below_fermi](#)
(page 1480)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

is_only_below_fermi

True if Delta is restricted to below fermi

See also:

[is_above_fermi](#) (page 1478), [is_below_fermi](#) (page 1479), [is_only_above_fermi](#) (page 1479)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

killable_index

Returns the index which is preferred to substitute in the final expression.

The index to substitute is the index with less information regarding fermi level. If indices contain same information, 'a' is preferred before 'b'.

See also:

[preferred_index](#) (page 1481)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
```

(continues on next page)

(continued from previous page)

```
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

preferred_index

Returns the index which is preferred to keep in the final expression.

The preferred index is the index with more information regarding fermi level. If indices contain same information, 'a' is preferred before 'b'.

See also:

[killable_index](#) (page 1480)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

class sympy.physics.secondquant.AnnihilateBoson

Bosonic annihilation operator.

Examples

```
>>> from sympy.physics.secondquant import B
>>> from sympy.abc import x
>>> B(x)
AnnihilateBoson(x)
```

apply_operator(state)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, BKet
>>> from sympy.abc import x, y, n
>>> B(x).apply_operator(y)
```

(continues on next page)

(continued from previous page)

```
y*AnnihilateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.CreateBoson

Bosonic creation operator.

apply_operator(state)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.AnnihilateFermion

Fermionic annihilation operator.

apply_operator(state)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

is_only_q_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_annihilator
True
>>> F(i).is_only_q_annihilator
False
>>> F(p).is_only_q_annihilator
False
```


is_only_q_creator

Always create a quasi-particle? (create hole or create particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_creator
False
>>> F(i).is_only_q_creator
True
>>> F(p).is_only_q_creator
False
```

is_q_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_annihilator
1
>>> F(i).is_q_annihilator
0
>>> F(p).is_q_annihilator
1
```

is_q_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_creator
0
>>> F(i).is_q_creator
-1
>>> F(p).is_q_creator
-1
```

class sympy.physics.secondquant.CreateFermion

Fermionic creation operator.

apply_operator(state)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y>CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

is_only_q_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_annihilator
False
>>> Fd(i).is_only_q_annihilator
True
>>> Fd(p).is_only_q_annihilator
False
```

is_only_q_creator

Always create a quasi-particle? (create hole or create particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_creator
True
>>> Fd(i).is_only_q_creator
False
>>> Fd(p).is_only_q_creator
False
```

is_q_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_annihilator
0
>>> Fd(i).is_q_annihilator
-1
```

(continues on next page)

(continued from previous page)

```
>>> Fd(p).is_q_annihilator
-1
```

is_q_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_creator
1
>>> Fd(i).is_q_creator
0
>>> Fd(p).is_q_creator
1
```

class sympy.physics.secondquant.FockState

Many particle Fock state with a sequence of occupation numbers.

Anywhere you can have a FockState, you can also have S.Zero. All code must check for this!

Base class to represent FockStates.

class sympy.physics.secondquant.FockStateBra

Representation of a bra.

class sympy.physics.secondquant.FockStateKet

Representation of a ket.

class sympy.physics.secondquant.FockStateBosonKet

Many particle Fock state with a sequence of occupation numbers.

Occupation numbers can be any integer ≥ 0 .

Examples

```
>>> from sympy.physics.secondquant import BKet
>>> BKet([1, 2])
FockStateBosonKet((1, 2))
```

class sympy.physics.secondquant.FockStateBosonBra

Describes a collection of BosonBra particles.

Examples

```
>>> from sympy.physics.secondquant import BBra
>>> BBra([1, 2])
FockStateBosonBra((1, 2))
```

`sympy.physics.secondquant.BBra`
alias of `sympy.physics.secondquant.FockStateBosonBra` (page 1485)

`sympy.physics.secondquant.BKet`
alias of `sympy.physics.secondquant.FockStateBosonKet` (page 1485)

`sympy.physics.secondquant.FBra`
alias of `sympy.physics.secondquant.FockStateFermionBra`

`sympy.physics.secondquant.FKet`
alias of `sympy.physics.secondquant.FockStateFermionKet`

`sympy.physics.secondquant.F`
alias of `sympy.physics.secondquant.AnnihilateFermion` (page 1482)

`sympy.physics.secondquant.Fd`
alias of `sympy.physics.secondquant.CreateFermion` (page 1483)

`sympy.physics.secondquant.B`
alias of `sympy.physics.secondquant.AnnihilateBoson` (page 1481)

`sympy.physics.secondquant.Bd`
alias of `sympy.physics.secondquant.CreateBoson` (page 1482)

`sympy.physics.secondquant.apply_operators(e)`
Take a sympy expression with operators and states and apply the operators.

Examples

```
>>> from sympy.physics.secondquant import apply_operators
>>> from sympy import sympify
>>> apply_operators(sympify(3)+4)
7
```

class `sympy.physics.secondquant.InnerProduct`

An unevaluated inner product between a bra and ket.

Currently this class just reduces things to a product of Kronecker Deltas. In the future, we could introduce abstract states like $|a\rangle$ and $|b\rangle$, and leave the inner product unevaluated as $\langle a|b\rangle$.

bra

Returns the bra part of the state

ket

Returns the ket part of the state

class `sympy.physics.secondquant.BosonicBasis`

Base class for a basis set of bosonic Fock states.

class `sympy.physics.secondquant.VarBosonicBasis(n_max)`

A single state, variable particle number basis set.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b
```

(continues on next page)

(continued from previous page)

```
[FockState((0,)), FockState((1,)), FockState((2,)),
 FockState((3,)), FockState((4,))]
```

index(state)

Returns the index of state in basis.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(3)
>>> state = b.state(1)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,))]
>>> state
FockStateBosonKet((1,))
>>> b.index(state)
1
```

state(i)

The state of a single basis.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b.state(3)
FockStateBosonKet((3,))
```

class sympy.physics.secondquant.FixedBosonicBasis(n_particles, n_levels)

Fixed particle number basis set.

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 2)
>>> state = b.state(1)
>>> b
[FockState((2, 0)), FockState((1, 1)), FockState((0, 2))]
>>> state
FockStateBosonKet((1, 1))
>>> b.index(state)
1
```

index(state)

Returns the index of state in basis.

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.index(b.state(3))
3
```

state(i)

Returns the state that lies at index i of the basis

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.state(3)
FockStateBosonKet((1, 0, 1))
```

class sympy.physics.secondquant.Commutator

The Commutator: $[A, B] = A*B - B*A$

The arguments are ordered according to `._cmp_()`

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import Commutator
>>> A, B = symbols('A,B', commutative=False)
>>> Commutator(B, A)
-Commutator(A, B)
```

Evaluate the commutator with `.doit()`

```
>>> comm = Commutator(A,B); comm
Commutator(A, B)
>>> comm.doit()
A*B - B*A
```

For two second quantization operators the commutator is evaluated immediately:

```
>>> from sympy.physics.secondquant import Fd, F
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> p,q = symbols('p,q')
```

```
>>> Commutator(Fd(a), Fd(i))
2*NO(CreateFermion(a)*CreateFermion(i))
```

But for more complicated expressions, the evaluation is triggered by a call to `.doit()`

```
>>> comm = Commutator(Fd(p)*Fd(q), Fd(i)); comm
Commutator(CreateFermion(p)*CreateFermion(q), AnnihilateFermion(i))
>>> comm.doit(wicks=True)
-KroneckerDelta(i, p)*CreateFermion(q) +
KroneckerDelta(i, q)*CreateFermion(p)
```

doit(hints)**

Enables the computation of complex expressions.

Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy import symbols
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> c = Commutator(Fd(a)*F(i), Fd(b)*F(j))
>>> c.doit(wicks=True)
0
```

classmethod eval(a, b)

The Commutator [A,B] is on canonical form if $A < B$.

Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy.abc import x
>>> c1 = Commutator(F(x), Fd(x))
>>> c2 = Commutator(Fd(x), F(x))
>>> Commutator.eval(c1, c2)
0
```

sympy.physics.secondquant.matrix_rep(op, basis)

Find the representation of an operator in a basis.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis, B, matrix_rep
>>> b = VarBosonicBasis(5)
>>> o = B(0)
>>> matrix_rep(o, b)
Matrix([
[0, 1,      0,      0, 0],
[0, 0, sqrt(2),      0, 0],
[0, 0,      0, sqrt(3), 0],
[0, 0,      0,      0, 2],
[0, 0,      0,      0, 0]])
```

sympy.physics.secondquant.contraction(a, b)

Calculates contraction of Fermionic operators a and b.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, Fd, contraction
>>> p, q = symbols('p,q')
>>> a, b = symbols('a,b', above_fermi=True)
>>> i, j = symbols('i,j', below_fermi=True)
```

A contraction is non-zero only if a quasi-creator is to the right of a quasi-annihilator:

```
>>> contraction(F(a),Fd(b))
KroneckerDelta(a, b)
>>> contraction(Fd(i),F(j))
KroneckerDelta(i, j)
```

For general indices a non-zero result restricts the indices to below/above the fermi surface:

```
>>> contraction(Fd(p),F(q))
KroneckerDelta(_i, q)*KroneckerDelta(p, q)
>>> contraction(F(p),Fd(q))
KroneckerDelta(_a, q)*KroneckerDelta(p, q)
```

Two creators or two annihilators always vanishes:

```
>>> contraction(F(p),F(q))
0
>>> contraction(Fd(p),Fd(q))
0
```

`sympy.physics.secondquant.wicks(e, **kw_args)`
Returns the normal ordered equivalent of an expression using Wicks Theorem.

Examples

```
>>> from sympy import symbols, Function, Dummy
>>> from sympy.physics.secondquant import wicks, F, Fd, NO
>>> p,q,r = symbols('p,q,r')
>>> wicks(Fd(p)*F(q))
d(p, q)*d(q, _i) + NO(CreateFermion(p)*AnnihilateFermion(q))
```

By default, the expression is expanded:

```
>>> wicks(F(p)*(F(q)+F(r)))
NO(AnnihilateFermion(p)*AnnihilateFermion(q)) + NO(
AnnihilateFermion(p)*AnnihilateFermion(r))
```

With the keyword 'keep_only_fully_contracted=True', only fully contracted terms are returned.

By request, the result can be simplified in the following order: - KroneckerDelta functions are evaluated - Dummy variables are substituted consistently across terms

```
>>> p, q, r = symbols('p q r', cls=Dummy)
>>> wicks(Fd(p)*(F(q)+F(r)), keep_only_fully_contracted=True)
KroneckerDelta(_i, _q)*KroneckerDelta(
    _p, _q) + KroneckerDelta(_i, _r)*KroneckerDelta(_p, _r)
```

class sympy.physics.secondquant.NO

This Object is used to represent normal ordering brackets.

i.e. {abcd} sometimes written :abcd:

Applying the function NO(arg) to an argument means that all operators in the argument will be assumed to anticommute, and have vanishing contractions. This allows an immediate reordering to canonical form upon object creation.


```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
>>> p,q = symbols('p,q')
>>> NO(Fd(p)*F(q))
NO(CreateFermion(p)*AnnihilateFermion(q))
>>> NO(F(q)*Fd(p))
-NO(CreateFermion(p)*AnnihilateFermion(q))
```

Note: If you want to generate a normal ordered equivalent of an expression, you should use the function `wicks()`. This class only indicates that all operators inside the brackets anticommute, and have vanishing contractions. Nothing more, nothing less.

doit(**kw_args)

Either removes the brackets or enables complex computations in its arguments.

Examples

```
>>> from sympy.physics.secondquant import NO, Fd, F
>>> from textwrap import fill
>>> from sympy import symbols, Dummy
>>> p,q = symbols('p,q', cls=Dummy)
>>> print(fill(str(NO(Fd(p)*F(q)).doit())))
KroneckerDelta(_a, _p)*KroneckerDelta(_a,
_q)*CreateFermion(_a)*AnnihilateFermion(_a) + KroneckerDelta(_a,
_p)*KroneckerDelta(_i, _q)*CreateFermion(_a)*AnnihilateFermion(_i) -
KroneckerDelta(_a, _q)*KroneckerDelta(_i,
_p)*AnnihilateFermion(_a)*CreateFermion(_i) - KroneckerDelta(_i,
_p)*KroneckerDelta(_i, _q)*AnnihilateFermion(_i)*CreateFermion(_i)
```

get_subNO(i)

Returns a NO() without FermionicOperator at index i.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, NO
>>> p,q,r = symbols('p,q,r')
```

```
>>> NO(F(p)*F(q)*F(r)).get_subNO(1)
NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

has_q_annihilators

Return 0 if the rightmost argument of the first argument is a not a q_annihilator, else 1 if it is above fermi or -1 if it is below fermi.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
```

```
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> N0(Fd(a)*Fd(i)).has_q_annihilators
-1
>>> N0(F(i)*F(a)).has_q_annihilators
1
>>> N0(Fd(a)*F(i)).has_q_annihilators
0
```

has_q_creators

Return 0 if the leftmost argument of the first argument is a not a `q_creator`, else 1 if it is above fermi or -1 if it is below fermi.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import N0, F, Fd
```

```
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> N0(Fd(a)*Fd(i)).has_q_creators
1
>>> N0(F(i)*F(a)).has_q_creators
-1
>>> N0(Fd(i)*F(a)).has_q_creators
0
```

iter_q_annihilators()

Iterates over the annihilation operators.

Examples

```
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import N0, F, Fd
>>> no = N0(Fd(a)*F(i)*F(b)*Fd(j))
```

```
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

iter_q_creators()

Iterates over the creation operators.

Examples

```
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a)*F(i)*F(b)*Fd(j))
```

```
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

`sympy.physics.secondquant.evaluate_deltas(e)`

We evaluate KroneckerDelta symbols in the expression assuming Einstein summation.

If one index is repeated it is summed over and in effect substituted with the other one. If both indices are repeated we substitute according to what is the preferred index. this is determined by `KroneckerDelta.preferred_index` and `KroneckerDelta.killable_index`.

In case there are no possible substitutions or if a substitution would imply a loss of information, nothing is done.

In case an index appears in more than one KroneckerDelta, the resulting substitution depends on the order of the factors. Since the ordering is platform dependent, the literal expression resulting from this function may be hard to predict.

Examples

We assume the following:

```
>>> from sympy import symbols, Function, Dummy, KroneckerDelta
>>> from sympy.physics.secondquant import evaluate_deltas
>>> i, j = symbols('i j', below_fermi=True, cls=Dummy)
>>> a, b = symbols('a b', above_fermi=True, cls=Dummy)
>>> p, q = symbols('p q', cls=Dummy)
>>> f = Function('f')
>>> t = Function('t')
```

The order of preference for these indices according to `KroneckerDelta` is (a, b, i, j, p, q).

Trivial cases:

```
>>> evaluate_deltas(KroneckerDelta(i,j)*f(i))      # d_ij f(i) -> f(j)
f(_j)
>>> evaluate_deltas(KroneckerDelta(i,j)*f(j))      # d_ij f(j) -> f(i)
f(_i)
>>> evaluate_deltas(KroneckerDelta(i,p)*f(p))      # d_ip f(p) -> f(i)
f(_i)
>>> evaluate_deltas(KroneckerDelta(q,p)*f(p))      # d_qp f(p) -> f(q)
f(_q)
>>> evaluate_deltas(KroneckerDelta(q,p)*f(q))      # d_qp f(q) -> f(p)
f(_p)
```

More interesting cases:

```
>>> evaluate_deltas(KroneckerDelta(i,p)*t(a,i)*f(p,q))
f(_i, _q)*t(_a, _i)
>>> evaluate_deltas(KroneckerDelta(a,p)*t(a,i)*f(p,q))
f(_a, _q)*t(_a, _i)
>>> evaluate_deltas(KroneckerDelta(p,q)*f(p,q))
f(_p, _p)
```

Finally, here are some cases where nothing is done, because that would imply a loss of information:

```
>>> evaluate_deltas(KroneckerDelta(i,p)*f(q))
f(_q)*KroneckerDelta(_i, _p)
>>> evaluate_deltas(KroneckerDelta(i,p)*f(i))
f(_i)*KroneckerDelta(_i, _p)
```

class `sympy.physics.secondquant.AntiSymmetricTensor`

Stores upper and lower indices in separate Tuple's.

Each group of indices is assumed to be antisymmetric.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (i, a), (b, j))
-AntiSymmetricTensor(v, (a, i), (b, j))
```

As you can see, the indices are automatically sorted to a canonical form.

doit(**kw_args)

Returns self.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j)).doit()
AntiSymmetricTensor(v, (a, i), (b, j))
```

lower

Returns the lower indices.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).lower
(b, j)
```

symbol

Returns the symbol of the tensor.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).symbol
v
```

upper

Returns the upper indices.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).upper
(a, i)
```

`sympy.physics.secondquant.substitute_dummies(expr, new_indices=False, pretty_indices={})`

Collect terms by substitution of dummy variables.

This routine allows simplification of Add expressions containing terms which differ only due to dummy variables.

The idea is to substitute all dummy variables consistently depending on the structure of the term. For each term, we obtain a sequence of all dummy variables, where the order is determined by the index range, what factors the index belongs to and its position in each factor. See `_get_ordered_dummies()` for more information about the sorting of dummies. The index sequence is then substituted consistently in each term.

Examples

```
>>> from sympy import symbols, Function, Dummy
>>> from sympy.physics.secondquant import substitute_dummies
>>> a,b,c,d = symbols('a b c d', above_fermi=True, cls=Dummy)
>>> i,j = symbols('i j', below_fermi=True, cls=Dummy)
>>> f = Function('f')
```

```
>>> expr = f(a,b) + f(c,d); expr
f(_a, _b) + f(_c, _d)
```

Since a, b, c and d are equivalent summation indices, the expression can be simplified to a single term (for which the dummy indices are still summed over)

```
>>> substitute_dummies(expr)
2*f(_a, _b)
```

Controlling output:

By default the dummy symbols that are already present in the expression will be reused in a different permutation. However, if `new_indices=True`, new dummies will be generated and inserted. The keyword `'pretty_indices'` can be used to control this generation of new symbols.

By default the new dummies will be generated on the form `i_1`, `i_2`, `a_1`, etc. If you supply a dictionary with key:value pairs in the form:

```
{ index_group: string_of_letters }
```

The letters will be used as labels for the new dummy symbols. The `index_groups` must be one of `'above'`, `'below'` or `'general'`.

```
>>> expr = f(a,b,i,j)
>>> my_dummies = { 'above':'st', 'below':'uv' }
>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_s, _t, _u, _v)
```

If we run out of letters, or if there is no keyword for some `index_group` the default dummy generator will be used as a fallback:

```
>>> p,q = symbols('p q', cls=Dummy) # general indices
>>> expr = f(p,q)
>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_p_0, _p_1)
```

class sympy.physics.secondquant.PermutationOperator

Represents the index permutation operator $P(ij)$.

$P(ij)*f(i)*g(j) = f(i)*g(j) - f(j)*g(i)$

get_permuted(expr)

Returns -expr with permuted indices.

```
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import PermutationOperator
>>> p,q = symbols('p,q')
>>> f = Function('f')
>>> PermutationOperator(p,q).get_permuted(f(p,q))
-f(q, p)
```

`sympy.physics.secondquant.simplify_index_permutations(expr, permutation_operators)`

Performs simplification by introducing PermutationOperators where appropriate.

Schematically: $[abij] - [abji] - [baij] + [baji] \rightarrow P(ab)P(ij)[abij]$

`permutation_operators` is a list of PermutationOperators to consider.

If `permutation_operators=[P(ab),P(ij)]` we will try to introduce the permutation operators $P(ij)$ and $P(ab)$ in the expression. If there are other possible simplifications, we ignore them.

```
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import simplify_index_permutations
>>> from sympy.physics.secondquant import PermutationOperator
>>> p,q,r,s = symbols('p,q,r,s')
>>> f = Function('f')
>>> g = Function('g')
```

```
>>> expr = f(p)*g(q) - f(q)*g(p); expr
f(p)*g(q) - f(q)*g(p)
>>> simplify_index_permutations(expr,[PermutationOperator(p,q)])
f(p)*g(q)*PermutationOperator(p, q)
```

```
>>> PermutList = [PermutationOperator(p,q),PermutationOperator(r,s)]
>>> expr = f(p,r)*g(q,s) - f(q,r)*g(p,s) + f(q,s)*g(p,r) - f(p,s)*g(q,r)
>>> simplify_index_permutations(expr,PermutList)
f(p, r)*g(q, s)*PermutationOperator(p, q)*PermutationOperator(r, s)
```

Wigner Symbols

Wigner, Clebsch-Gordan, Racah, and Gaunt coefficients

Collection of functions for calculating Wigner 3j, 6j, 9j, Clebsch-Gordan, Racah as well as Gaunt coefficients exactly, all evaluating to a rational number times the square root of a rational number [Rasch03] (page 1872).

Please see the description of the individual functions for further details and examples.

References

Credits and Copyright

This code was taken from Sage with the permission of all authors:

<https://groups.google.com/forum/#!topic/sage-devel/M4NZdu-7O38>

AUTHORS:

- Jens Rasch (2009-03-24): initial version for Sage
- Jens Rasch (2009-05-31): updated to sage-4.0

Copyright (C) 2008 Jens Rasch <jyr2000@gmail.com>

`sympy.physics.wigner.clebsch_gordan(j_1, j_2, j_3, m_1, m_2, m_3)`

Calculates the Clebsch-Gordan coefficient $\langle j_1 m_1 j_2 m_2 | j_3 m_3 \rangle$.

The reference for this function is [Edmonds74] (page 1872).

INPUT:

- $j_1, j_2, j_3, m_1, m_2, m_3$ - integer or half integer

OUTPUT:

Rational number times the square root of a rational number.

EXAMPLES:

```
>>> from sympy import S
>>> from sympy.physics.wigner import clebsch_gordan
>>> clebsch_gordan(S(3)/2, S(1)/2, 2, S(3)/2, S(1)/2, 2)
1
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, S(3)/2, -S(1)/2, 1)
sqrt(3)/2
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, -S(1)/2, S(1)/2, 0)
-sqrt(2)/2
```

NOTES:

The Clebsch-Gordan coefficient will be evaluated via its relation to Wigner 3j symbols:

$$\langle j_1 m_1 j_2 m_2 | j_3 m_3 \rangle = (-1)^{j_1 - j_2 + m_3} \sqrt{2j_3 + 1} \text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, -m_3)$$

See also the documentation on Wigner 3j symbols which exhibit much higher symmetry relations than the Clebsch-Gordan coefficient.

AUTHORS:

- Jens Rasch (2009-03-24): initial version

`sympy.physics.wigner.dot_rot_grad_Ynm(j, p, l, m, theta, phi)`

Returns dot product of rotational gradients of spherical harmonics.

This function returns the right hand side of the following expression:

$$\vec{R}Y_j^p \cdot \vec{R}Y_l^m = (-1)^{m+p} \sum_{k=|l-j|}^{l+j} Y_k^{m+p} * \alpha_{l,m,j,p,k} * \frac{1}{2}(k^2 - j^2 - l^2 + k - j - l)$$

Arguments

j, p, l, m indices in spherical harmonics (expressions or integers) θ, ϕ angle arguments in spherical harmonics

Example

```
>>> from sympy import symbols
>>> from sympy.physics.wigner import dot_rot_grad_Ynm
>>> theta, phi = symbols("theta phi")
>>> dot_rot_grad_Ynm(3, 2, 2, 0, theta, phi).doit()
3*sqrt(55)*Ynm(5, 2, theta, phi)/(11*sqrt(pi))
```

`sympy.physics.wigner.gaunt(l_1, l_2, l_3, m_1, m_2, m_3, prec=None)`

Calculate the Gaunt coefficient.

The Gaunt coefficient is defined as the integral over three spherical harmonics:

$$\begin{aligned}\text{Gaunt}(l_1, l_2, l_3, m_1, m_2, m_3) &= \int Y_{l_1, m_1}(\Omega) Y_{l_2, m_2}(\Omega) Y_{l_3, m_3}(\Omega) d\Omega \\ &= \sqrt{\frac{(2l_1 + 1)(2l_2 + 1)(2l_3 + 1)}{4\pi}} \text{Wigner3j}(l_1, l_2, l_3, 0, 0, 0) \text{Wigner3j}(l_1, l_2, l_3, m_1, m_2, m_3)\end{aligned}$$

INPUT:

- `l_1, l_2, l_3, m_1, m_2, m_3` - integer
- `prec` - precision, default: None. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import gaunt
>>> gaunt(1,0,1,1,0,-1)
-1/(2*sqrt(pi))
>>> gaunt(1000,1000,1200,9,3,-12).n(64)
0.00689500421922113448...
```

It is an error to use non-integer values for l and m :

```
sage: gaunt(1.2,0,1.2,0,0,0)
Traceback (most recent call last):
...
ValueError: l values must be integer
sage: gaunt(1,0,1,1.1,0,-1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer
```

NOTES:

The Gaunt coefficient obeys the following symmetry rules:

- invariant under any permutation of the columns

$$\begin{aligned}Y(l_1, l_2, l_3, m_1, m_2, m_3) &= Y(l_3, l_1, l_2, m_3, m_1, m_2) \\ &= Y(l_2, l_3, l_1, m_2, m_3, m_1) \\ &= Y(l_3, l_2, l_1, m_3, m_2, m_1) \\ &= Y(l_1, l_3, l_2, m_1, m_3, m_2) \\ &= Y(l_2, l_1, l_3, m_2, m_1, m_3)\end{aligned}$$

- invariant under space inflection, i.e.

$$Y(l_1, l_2, l_3, m_1, m_2, m_3) = Y(l_1, l_2, l_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 Regge symmetries as inherited for the $3j$ symbols [\[Regge58\]](#) (page 1872)

- zero for l_1, l_2, l_3 not fulfilling triangle relation
- zero for violating any one of the conditions: $l_1 \geq |m_1|, l_2 \geq |m_2|, l_3 \geq |m_3|$
- non-zero only for an even sum of the l_i , i.e. $L = l_1 + l_2 + l_3 = 2n$ for n in $\mathbb{N}$

ALGORITHM:

This function uses the algorithm of [Liberatodebrito82] (page 1872) to calculate the value of the Gaunt coefficient exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03] (page 1872).

REFERENCES:

AUTHORS:

- Jens Rasch (2009-03-24): initial version for Sage

`sympy.physics.wigner.racah(aa, bb, cc, dd, ee, ff, prec=None)`

Calculate the Racah symbol $W(a, b, c, d; e, f)$.

INPUT:

- `a, ..., f` - integer or half integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import racah
>>> racah(3,3,3,3,3,3)
-1/14
```

NOTES:

The Racah symbol is related to the Wigner 6j symbol:

$$\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

Please see the 6j symbol for its much richer symmetries and for additional properties.

ALGORITHM:

This function uses the algorithm of [Edmonds74] (page 1872) to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03] (page 1872).

AUTHORS:

- Jens Rasch (2009-03-24): initial version

`sympy.physics.wigner.wigner_3j(j_1, j_2, j_3, m_1, m_2, m_3)`

Calculate the Wigner 3j symbol $Wigner3j(j_1, j_2, j_3, m_1, m_2, m_3)$.

INPUT:

- `j_1, j_2, j_3, m_1, m_2, m_3` - integer or half integer

OUTPUT:

Rational number times the square root of a rational number.

Examples

```
>>> from sympy.physics.wigner import wigner_3j
>>> wigner_3j(2, 6, 4, 0, 0, 0)
sqrt(715)/143
>>> wigner_3j(2, 6, 4, 0, 0, 1)
0
```

It is an error to have arguments that are not integer or half integer values:

```
sage: wigner_3j(2.1, 6, 4, 0, 0, 0)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer
sage: wigner_3j(2, 6, 4, 1, 0, -1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer or half integer
```

NOTES:

The Wigner 3j symbol obeys the following symmetry rules:

- invariant under any permutation of the columns (with the exception of a sign change where $J := j_1 + j_2 + j_3$):

$$\begin{aligned} \text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) &= \text{Wigner3j}(j_3, j_1, j_2, m_3, m_1, m_2) \\ &= \text{Wigner3j}(j_2, j_3, j_1, m_2, m_3, m_1) \\ &= (-1)^J \text{Wigner3j}(j_3, j_2, j_1, m_3, m_2, m_1) \\ &= (-1)^J \text{Wigner3j}(j_1, j_3, j_2, m_1, m_3, m_2) \\ &= (-1)^J \text{Wigner3j}(j_2, j_1, j_3, m_2, m_1, m_3) \end{aligned}$$

- invariant under space inflection, i.e.

$$\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) = (-1)^J \text{Wigner3j}(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 additional symmetries based on the work by [\[Regge58\]](#) (page 1872)
- zero for j_1, j_2, j_3 not fulfilling triangle relation
- zero for $m_1 + m_2 + m_3 \neq 0$
- zero for violating any one of the conditions $j_1 \geq |m_1|$, $j_2 \geq |m_2|$, $j_3 \geq |m_3|$

ALGORITHM:

This function uses the algorithm of [\[Edmonds74\]](#) (page 1872) to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [\[Rasch03\]](#) (page 1872).

REFERENCES:

AUTHORS:

- Jens Rasch (2009-03-24): initial version

`sympy.physics.wigner.wigner_6j(j_1, j_2, j_3, j_4, j_5, j_6, prec=None)`

Calculate the Wigner 6j symbol $\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6)$.

INPUT:

- `j_1, ..., j_6` - integer or half integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import wigner_6j
>>> wigner_6j(3,3,3,3,3,3)
-1/14
>>> wigner_6j(5,5,5,5,5,5)
1/52
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_6j(2.5,2.5,2.5,2.5,2.5,2.5)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle_
↪ relation
sage: wigner_6j(0.5,0.5,1.1,0.5,0.5,1.1)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle_
↪ relation
```

NOTES:

The Wigner 6j symbol is related to the Racah symbol but exhibits more symmetries as detailed below.

$$\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

The Wigner 6j symbol obeys the following symmetry rules:

- Wigner 6j symbols are left invariant under any permutation of the columns:

$$\begin{aligned} \text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) &= \text{Wigner6j}(j_3, j_1, j_2, j_6, j_4, j_5) \\ &= \text{Wigner6j}(j_2, j_3, j_1, j_5, j_6, j_4) \\ &= \text{Wigner6j}(j_3, j_2, j_1, j_6, j_5, j_4) \\ &= \text{Wigner6j}(j_1, j_3, j_2, j_4, j_6, j_5) \\ &= \text{Wigner6j}(j_2, j_1, j_3, j_5, j_4, j_6) \end{aligned}$$

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

$$\text{Wigner6j}(j_1, j_2, j_3, j_4, j_5, j_6) = \text{Wigner6j}(j_1, j_5, j_6, j_4, j_2, j_3) = \text{Wigner6j}(j_4, j_2, j_6, j_1, j_5, j_3) = \text{Wigner6j}(j_4, j_5,$$

- additional 6 symmetries [Regge59] (page 1872) giving rise to 144 symmetries in total
- only non-zero if any triple of j 's fulfill a triangle relation

ALGORITHM:

This function uses the algorithm of [Edmonds74] (page 1872) to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03] (page 1872).

REFERENCES:

`sympy.physics.wigner.wigner_9j(j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, prec=None)`

Calculate the Wigner 9j symbol $Wigner9j(j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9)$.

INPUT:

- $j_1, \dots, j_9$ - integer or half integer
- `prec` - precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import wigner_9j
>>> wigner_9j(1,1,1, 1,1,1, 1,1,0 ,prec=64) # ==1/18
0.05555555...
```

```
>>> wigner_9j(1/2,1/2,0, 1/2,3/2,1, 0,1,1 ,prec=64) # ==1/6
0.1666666...
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_9j(0.5,0.5,0.5, 0.5,0.5,0.5, 0.5,0.5,0.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle_
↪relation
sage: wigner_9j(1,1,1, 0.5,1,1.5, 0.5,1,2.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle_
↪relation
```

ALGORITHM:

This function uses the algorithm of [Edmonds74] (page 1872) to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03] (page 1872).

Unit systems

This module integrates unit systems into SymPy, allowing a user choose which system to use when doing their computations and providing utilities to display and convert units.

Unit systems are composed of units and constants, which are themselves described from dimensions and numbers, and possibly a prefix. Quantities are defined by their unit and their numerical value, with respect to the current system.

The main advantage of this implementation over the old unit module is that it divides the units in unit systems, so that the user can decide which units to use, instead of having all in the name space (for example astrophysicists can only use units with ua, Earth or Sun masses, the theoreticists will use natural system, etc.). Moreover it allows a better control over the dimensions and conversions.

Ideas about future developments can be found on the [Github wiki](#).

Philosophy behind unit systems

Dimensions

Introduction

At the root of unit systems are dimension systems, whose structure mainly determines the one of unit systems. Our definition could seem rough but they are largely sufficient for our purposes.

A dimension will be defined as a property which is measurable and assigned to a specific phenomenon. In this sense dimensions are different from pure numbers because they carry some extra-sense, and for this reason two different dimensions cannot be added. For example time or length are dimensions, but also any other things which has some sense for us, like angle, number of particles (moles...) or information (bits...).

From this point of view the only truly dimensionless quantity are pure numbers. The idea of being dimensionless is very system-dependent, as can be seen from the $(c, \hbar, G)$, in which all units appears to be dimensionless in the usual common sense. This is unavoidable for computability of generic unit systems (but at the end we can tell the program what is dimensionless).

Dimensions can be composed together by taking their product or their ratio (to be defined below). For example the velocity is defined as length divided by time, or we can see the length as velocity multiplied by time, depending of what we see as the more fundamental: in general we can select a set of base dimensions from which we can describe all the others.

Group structure

After this short introduction whose aim was to introduce the dimensions from an intuitive perspective, we describe the mathematical structure. A dimension system with n independent dimensions $\{d_i\}_{i=1,\dots,n}$ is described by a multiplicative group G :

- there an identity element 1 corresponding to pure numbers;
- the product $D_3 = D_1 D_2$ of two elements $D_1, D_2 \in G$ is also in G ;
- any element $D \in G$ has an inverse $D^{-1} \in G$.

We denote

$$D^n = \underbrace{D \times \cdots \times D}_{n \text{ times}},$$

and by definition $D^0 = 1$. The $\{d_i\}_{i=1,\dots,n}$ are called generators of the group since any element $D \in G$ can be expressed as the product of powers of the generators:

$$D = \prod_{i=1}^n d_i^{a_i}, \quad a_i \in \mathbf{Z}.$$

The identity is given for $a_i = 0, \forall i$, while we recover the generator d_i for $a_i = 1, a_j = 0, \forall j \neq i$. This group has the following properties:

1. abelian, since the generator commutes, $[d_i, d_j] = 0$;
2. countable (infinite but discrete) since the elements are indexed by the powers of the generators¹.

One can change the dimension basis $\{d'_i\}_{i=1,\dots,n}$ by taking some combination of the old generators:

$$d'_i = \prod_{j=1}^n d_j^{P_{ij}}.$$

Linear space representation

It is possible to use the linear space $\mathbf{Z}^n$ as a representation of the group since the power coefficients a_i carry all the information one needs (we do not distinguish between the element of the group and its representation):

$$(d_i)_j = \delta_{ij}, \quad D = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}.$$

The change of basis to d'_i follows the usual rule of change of basis for linear space, the matrix being given by the coefficients P_{ij} , which are simply the coefficients of the new vectors in term of the old basis:

$$d'_i = P_{ij} d_j.$$

We will use this last solution in our algorithm.

An example

In order to illustrate all this formalism, we end this section with a specific example, the MKS system (m, kg, s) with dimensions (L: length, M: mass, T: time). They are represented as (we will always sort the vectors in alphabetic order)

$$L = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad M = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

¹ In general we will consider only dimensions with a maximum coefficient, so we can only a truncation of the group; but this is not useful for the algorithm.

Other dimensions can be derived, for example velocity V or action A

$$V = LT^{-1}, \quad A = ML^2T^{-2},$$
$$V = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \quad A = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}.$$

We can change the basis to go to the natural system $(m, c, \hbar)$ with dimension (L: length, V: velocity, A: action)². In this basis the generators are

$$A = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad L = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad V = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

whereas the mass and time are given by

$$T = LV^{-1}, \quad M = AV^{-2},$$
$$T = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \quad M = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}.$$

Finally the inverse change of basis matrix P^{-1} is obtained by gluing the vectors expressed in the old basis:

$$P^{-1} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ -2 & 0 & -1 \end{pmatrix}.$$

To find the change of basis matrix we just have to take the inverse

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & -2 & -1 \end{pmatrix}.$$

Quantities

A quantity is defined by its name, dimension and factor to a canonical quantity of the same dimension. The canonical quantities are an internal reference of the units module and should not be relevant for end-users. Both units and physical constants are quantities.

Units

Units, such as meters, seconds and kilograms, are usually reference quantities chosen by men to refer to other quantities.

After defining several units of different dimensions we can form a unit system, which is basically a dimension system with a notion of scale.

Constants

Physical constants are just quantities. They indicate that we used not to understand that two dimensions are in fact the same. For example, we see a velocity for the light different from

² We anticipate a little by considering c and $\hbar$ as units and not as physical constants.

1 because we do not think that time is the same as space (which is normal because of our sense; but it is different at the fundamental level). For example, once there was the “heat constant” which allowed to convert between joules and calories since people did not know that heat was energy. As soon as they understood it they fixed this constant to 1 (this is a very schematic story).

We can interpret the fact that now we fix the value of fundamental constants in the SI as showing that they are units (and we use them to define the other usual units).

The need for a reference

It is not possible to define from scratch units and unit systems: one needs to define some references, and then build the rest over them. Said in another way, we need an origin for the scales of our units (i.e. a unit with factor 1), and to be sure that all units of a given dimension are defined consistently we need to use the same origin for all of them. This can happen if we want to use a derived unit as a base units in another system: we should not define it as having a scale 1, because, even if it is inconsistent inside the system, we could not convert to the first system since we have two different units (from our point of view) of same scale (which means they are equal for the computer).

We will say that the dimensions and scales defined outside systems are canonical, because we use them for all computations. On the other side the dimensions and scales obtained with reference to a system are called physical, because they ultimately carry a sense.

Let’s use a concrete (and important) example: the case of the mass units. We would like to define the gram as the origin. We would like to define the gram as the canonical origin for the mass, so we assign it a scale 1. Then we can define a system (e.g. in chemistry) that take it as a base unit. The MKS system prefers to use the kilogram; a naive choice would be to attribute it a scale of 1 since it is a base, but we see that we could not convert to the chemistry system because g and kg have both been given the same factor. So we need to define kg as 1000 g, and only then use it as a base in MKS. But as soon as we ask the question “what is the factor of kg in MKS?”, we get the answer 1, since it is a base unit.

Thus we will define all computations without referring to a system, and it is only at the end that we can plug the result into a system to give the context we are interested in.

Literature

Examples

In the following sections we give few examples of what can be do with this module.

Dimensional analysis

We will start from the second Newton’s law

$$ma = F$$

where m , a and F are the mass, the acceleration and the force respectively. Knowing the dimensions of m (M) and a (LT^{-2}), we will determine the dimension of F ; obviously we will find that it is a force: MLT^{-2} .

From there we will use the expression of the gravitational force between the particle of mass m and the body of mass M , at a distance r

$$F = \frac{GmM}{r^2}$$

to determine the dimension of the Newton's constant G . The result should be $L^3M^{-1}T^{-2}$.

```
>>> from sympy import symbols
>>> from sympy.physics.units import length, mass, acceleration, force
>>> from sympy.physics.units import gravitational_constant as G
>>> from sympy.physics.units.dimensions import dimsys_SI
>>> F = mass*acceleration
>>> F
Dimension(acceleration*mass)
>>> dimsys_SI.get_dimensional_dependencies(F)
{'length': 1, 'mass': 1, 'time': -2}
>>> dimsys_SI.get_dimensional_dependencies(force)
{'length': 1, 'mass': 1, 'time': -2}
```

Dimensions cannot be compared directly, even if in the SI convention they are the same:

```
>>> F == force
False
```

Dimension system objects provide a way to test the equivalence of dimensions:

```
>>> dimsys_SI.equivalent_dims(F, force)
True
```

```
>>> m1, m2, r = symbols("m1 m2 r")
>>> grav_eq = G * m1 * m2 / r**2
>>> F2 = grav_eq.subs({m1: mass, m2: mass, r: length, G: G.dimension})
>>> F2
Dimension(mass*length*time**-2)
>>> F2.get_dimensional_dependencies()
{'length': 1, 'mass': 1, 'time': -2}
```

Note that one should first solve the equation, and then substitute with the dimensions.

Equation with quantities

Using Kepler's third law

$$\frac{T^2}{a^3} = \frac{4\pi^2}{GM}$$

we can find the Venus orbital period using the known values for the other variables (taken from Wikipedia). The result should be 224.701 days.

```
>>> from sympy import solve, symbols, pi, Eq
>>> from sympy.physics.units import Quantity, length, mass
>>> from sympy.physics.units import day, gravitational_constant as G
>>> from sympy.physics.units import meter, kilogram
>>> T = symbols("T")
>>> a = Quantity("venus_a")
```

Specify the dimension and scale in SI units:

```
>>> a.set_dimension(length, "SI")
>>> a.set_scale_factor(108208000e3*meter, "SI")
```

Add the solar mass as quantity:

```
>>> M = Quantity("solar_mass")
>>> M.set_dimension(mass, "SI")
>>> M.set_scale_factor(1.9891e30*kilogram, "SI")
```

Now Kepler's law:

```
>>> eq = Eq(T**2 / a**3, 4*pi**2 / G / M)
>>> eq
Eq(T**2/venus_a**3, 4*pi**2/(gravitational_constant*solar_mass))
>>> q = solve(eq, T)[1]
>>> q
2*pi*venus_a**(3/2)/(sqrt(gravitational_constant)*sqrt(solar_mass))
```

To convert to days, use the `convert_to` function (and possibly approximate the outcoming result):

```
>>> from sympy.physics.units import convert_to
>>> convert_to(q, day)
71.5123904642338*pi*day
>>> convert_to(q, day).n()
224.662800523082*day
```

We could also have the solar mass and the day as units coming from the astrophysical system, but I wanted to show how to create a unit that one needs.

We can see in this example that intermediate dimensions can be ill-defined, such as $\sqrt{G}$, but one should check that the final result - when all dimensions are combined - is well defined.

Dimensions and dimension systems

Definition of physical dimensions.

Unit systems will be constructed on top of these dimensions.

Most of the examples in the doc use MKS system and are presented from the computer point of view: from a human point, adding length to time is not legal in MKS but it is in natural system; for a computer in natural system there is no time dimension (but a velocity dimension instead) - in the basis - so the question of adding time to length has no meaning.

class `sympy.physics.units.dimensions.Dimension`

This class represent the dimension of a physical quantities.

The `Dimension` constructor takes as parameters a name and an optional symbol.

For example, in classical mechanics we know that time is different from temperature and dimensions make this difference (but they do not provide any measure of these quantites).

```
>>> from sympy.physics.units import Dimension
>>> length = Dimension('length')
>>> length
Dimension(length)
>>> time = Dimension('time')
```

(continues on next page)

(continued from previous page)

```
>>> time
Dimension(time)
```

Dimensions can be composed using multiplication, division and exponentiation (by a number) to give new dimensions. Addition and subtraction is defined only when the two objects are the same dimension.

```
>>> velocity = length / time
>>> velocity
Dimension(length/time)
```

It is possible to use a dimension system object to get the dimensional dependencies of a dimension, for example the dimension system used by the SI units convention can be used:

```
>>> from sympy.physics.units.dimensions import dimsys_SI
>>> dimsys_SI.get_dimensional_dependencies(velocity)
{'length': 1, 'time': -1}
>>> length + length
Dimension(length)
>>> l2 = length**2
>>> l2
Dimension(length**2)
>>> dimsys_SI.get_dimensional_dependencies(l2)
{'length': 2}
```

has_integer_powers(dim_sys)

Check if the dimension object has only integer powers.

All the dimension powers should be integers, but rational powers may appear in intermediate steps. This method may be used to check that the final result is well-defined.

is_dimensionless

Check if the dimension object really has a dimension.

A dimension should have at least one component with non-zero power.

class sympy.physics.units.dimensions.DimensionSystem

DimensionSystem represents a coherent set of dimensions.

The constructor takes three parameters:

- base dimensions;
- derived dimensions: these are defined in terms of the base dimensions (for example velocity is defined from the division of length by time);
- dependency of dimensions: how the derived dimensions depend on the base dimensions.

Optionally either the `derived_dims` or the `dimensional_dependencies` may be omitted.

can_transf_matrix

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Return the canonical transformation matrix from the canonical to the base dimension basis.

It is the inverse of the matrix computed with `inv_can_transf_matrix()`.

dim

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Give the dimension of the system.

That is return the number of dimensions forming the basis.

dim_can_vector(dim)

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Dimensional representation in terms of the canonical base dimensions.

dim_vector(dim)

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Vector representation in terms of the base dimensions.

inv_can_transf_matrix

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Compute the inverse transformation matrix from the base to the canonical dimension basis.

It corresponds to the matrix where columns are the vector of base dimensions in canonical basis.

This matrix will almost never be used because dimensions are always defined with respect to the canonical basis, so no work has to be done to get them in this basis. Nonetheless if this matrix is not square (or not invertible) it means that we have chosen a bad basis.

is_consistent

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Check if the system is well defined.

is_dimensionless(dimension)

Check if the dimension object really has a dimension.

A dimension should have at least one component with non-zero power.

list_can_dims

Useless method, kept for compatibility with previous versions.

DO NOT USE.

List all canonical dimension names.

print_dim_base(dim)

Give the string expression of a dimension in term of the basis symbols.

static sort_dims(dims)

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Sort dimensions given in argument using their str function.

This function will ensure that we get always the same tuple for a given set of dimensions.

Unit prefixes

Module defining unit prefixe class and some constants.

Constant dict for SI and binary prefixes are defined as `PREFIXES` and `BIN_PREFIXES`.

class `sympy.physics.units.prefixes.Prefix`

This class represent prefixes, with their name, symbol and factor.

Prefixes are used to create derived units from a given unit. They should always be encapsulated into units.

The factor is constructed from a base (default is 10) to some power, and it gives the total multiple or fraction. For example the kilometer km is constructed from the meter (factor 1) and the kilo (10 to the power 3, i.e. 1000). The base can be changed to allow e.g. binary prefixes.

A prefix multiplied by something will always return the product of this other object times the factor, except if the other object:

- is a prefix and they can be combined into a new prefix;
- defines multiplication with prefixes (which is the case for the `Unit` class).

Units and unit systems

Unit system for physical quantities; include definition of constants.

class `sympy.physics.units.unitsystem.UnitSystem`(base, units=(), name="", descr="")

`UnitSystem` represents a coherent set of units.

A unit system is basically a dimension system with notions of scales. Many of the methods are defined in the same way.

It is much better if all base units have a symbol.

dim

Give the dimension of the system.

That is return the number of units forming the basis.

extend(base, units=(), name="", description="")

Extend the current system into a new one.

Take the base and normal units of the current system to merge them to the base and normal units given in argument. If not provided, name and description are overridden by empty strings.

is_consistent

Check if the underlying dimension system is consistent.

print_unit_base(unit)

Useless method.

DO NOT USE, use instead `convert_to`.

Give the string expression of a unit in term of the basis.

Units are displayed by decreasing power.

Physical quantities

Physical quantities.

class `sympy.physics.units.quantities.Quantity`

Physical quantity: can be a unit of measure, a constant or a generic quantity.

abbrev

Symbol representing the unit name.

Prepend the abbreviation with the prefix symbol if it is defines.

convert_to(other)

Convert the quantity to another quantity of same dimensions.

Examples

```
>>> from sympy.physics.units import speed_of_light, meter, second
>>> speed_of_light
speed_of_light
>>> speed_of_light.convert_to(meter/second)
299792458*meter/second
```

```
>>> from sympy.physics.units import liter
>>> liter.convert_to(meter**3)
meter**3/1000
```

free_symbols

Return free symbols from quantity.

scale_factor

Overall magnitude of the quantity as compared to the canonical units.

Conversion between quantities

Several methods to simplify expressions involving unit objects.

`sympy.physics.units.util.convert_to(expr, target_units)`

Convert `expr` to the same expression with all of its units and quantities represented as factors of `target_units`, whenever the dimension is compatible.

`target_units` may be a single unit/quantity, or a collection of units/quantities.

Examples

```
>>> from sympy.physics.units import speed_of_light, meter, gram, second, day
>>> from sympy.physics.units import mile, newton, kilogram, atomic_mass_constant
>>> from sympy.physics.units import kilometer, centimeter
>>> from sympy.physics.units import convert_to
>>> convert_to(mile, kilometer)
25146*kilometer/15625
>>> convert_to(mile, kilometer).n()
1.609344*kilometer
>>> convert_to(speed_of_light, meter/second)
```

(continues on next page)

(continued from previous page)

```

299792458*meter/second
>>> convert_to(day, second)
86400*second
>>> 3*newton
3*newton
>>> convert_to(3*newton, kilogram*meter/second**2)
3*kilogram*meter/second**2
>>> convert_to(atomic_mass_constant, gram)
1.66053904e-24*gram

```

Conversion to multiple units:

```

>>> convert_to(speed_of_light, [meter, second])
299792458*meter/second
>>> convert_to(3*newton, [centimeter, gram, second])
300000*centimeter*gram/second**2

```

Conversion to Planck units:

```

>>> from sympy.physics.units import gravitational_constant, hbar
>>> convert_to(atomic_mass_constant, [gravitational_constant, speed_of_light,
↪hbar]).n()
7.62950196312651e-20*gravitational_constant**(-0.5)*hbar**0.5*speed_of_light**0.5

```

High energy physics

Abstract

Contains docstrings for methods in high energy physics.

Gamma matrices

Module to handle gamma matrices expressed as tensor objects.

Examples

```

>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G, LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
>>> G(i)
GammaMatrix(i)

```

Note that there is already an instance of GammaMatrixHead in four dimensions: GammaMatrix, which is simply declare as

```

>>> from sympy.physics.hep.gamma_matrices import GammaMatrix
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
>>> GammaMatrix(i)
GammaMatrix(i)

```


To access the metric tensor

```
>>> LorentzIndex.metric
metric(LorentzIndex, LorentzIndex)
```

`sympy.physics.hep.gamma_matrices.extract_type_tens(expression, component)`
Extract from a TensExpr all tensors with *component*.

Returns two tensor expressions:

- the first contains all Tensor of having *component*.
- the second contains all remaining.

`sympy.physics.hep.gamma_matrices.gamma_trace(t)`
trace of a single line of gamma matrices

Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,      gamma_
    trace, LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices, tensorhead
>>> p, q = tensorhead('p, q', [LorentzIndex], [[1]])
>>> i0,i1,i2,i3,i4,i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> gamma_trace(G(i0)*G(i1))
4*metric(i0, i1)
>>> gamma_trace(ps*ps) - 4*p(i0)*p(-i0)
0
>>> gamma_trace(ps*qs + ps*ps) - 4*p(i0)*p(-i0) - 4*p(i0)*q(-i0)
0
```

`sympy.physics.hep.gamma_matrices.kahane_simplify(expression)`

This function cancels contracted elements in a product of four dimensional gamma matrices, resulting in an expression equal to the given one, without the contracted gamma matrices.

Parameters 'expression' the tensor expression containing the gamma matrices to simplify.

Notes

If spinor indices are given, the matrices must be given in the order given in the product.

References

[1] Algorithm for Reducing Contracted Products of gamma Matrices, Joseph Kahane, Journal of Mathematical Physics, Vol. 9, No. 10, October 1968.

Examples

When using, always remember that the original expression coefficient has to be handled separately

```

>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G, LorentzIndex
>>> from sympy.physics.hep.gamma_matrices import kahane_simplify
>>> from sympy.tensor.tensor import tensor_indices
>>> i0, i1, i2 = tensor_indices('i0:3', LorentzIndex)
>>> ta = G(i0)*G(-i0)
>>> kahane_simplify(ta)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
>>> tb = G(i0)*G(i1)*G(-i0)
>>> kahane_simplify(tb)
-2*GammaMatrix(i1)
>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])

```

If there are no contractions, the same expression is returned

```

>>> tc = G(i0)*G(i1)
>>> kahane_simplify(tc)
GammaMatrix(i0)*GammaMatrix(i1)

```

Algorithm

The idea behind the algorithm is to use some well-known identities, i.e., for contractions enclosing an even number of γ matrices

$$\gamma^\mu \gamma_{a_1} \cdots \gamma_{a_{2N}} \gamma_\mu = 2(\gamma_{a_{2N}} \gamma_{a_1} \cdots \gamma_{a_{2N-1}} + \gamma_{a_{2N-1}} \cdots \gamma_{a_1} \gamma_{a_{2N}})$$

for an odd number of γ matrices

$$\gamma^\mu \gamma_{a_1} \cdots \gamma_{a_{2N+1}} \gamma_\mu = -2\gamma_{a_{2N+1}} \gamma_{a_{2N}} \cdots \gamma_{a_1}$$

Instead of repeatedly applying these identities to cancel out all contracted indices, it is possible to recognize the links that would result from such an operation, the problem is thus reduced to a simple rearrangement of free gamma matrices.

`sympy.physics.hep.gamma_matrices.simplify_gpgp(ex, sort=True)`
 simplify products $G(i)*p(-i)*G(j)*p(-j) \rightarrow p(i)*p(-i)$

Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
↳ LorentzIndex, simplify_gpgp
>>> from sympy.tensor.tensor import tensor_indices, tensorhead
>>> p, q = tensorhead('p, q', [LorentzIndex], [[1]])
>>> i0,i1,i2,i3,i4,i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> simplify_gpgp(ps*qs*qs)
GammaMatrix(-L_0)*p(L_0)*q(L_1)*q(-L_1)
```

The Physics Vector Module

Abstract

In this documentation the components of the `sympy.physics.vector` module have been discussed. `vector` has been written to facilitate the operations pertaining to 3-dimensional vectors, as functions of time or otherwise, in `sympy.physics`.

References for Physics/Vector

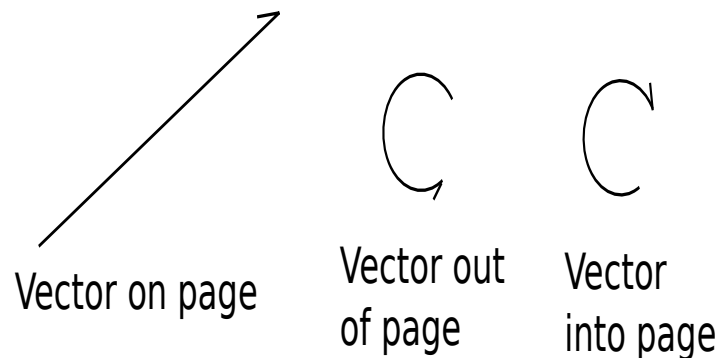
Guide to Vector

Vector & ReferenceFrame

In `vector`, vectors and reference frames are the “building blocks” of dynamic systems. This document will describe these mathematically and describe how to use them with this module’s code.

Vector

A vector is a geometric object that has a magnitude (or length) and a direction. Vectors in 3-space are often represented on paper as:



Vector Algebra

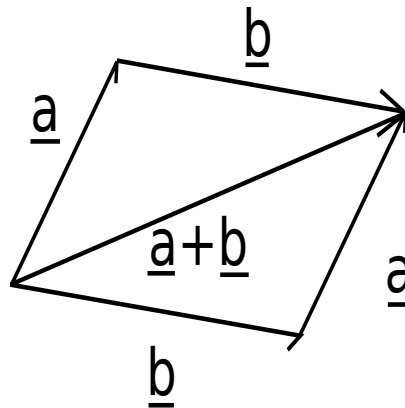
Vector algebra is the first topic to be discussed.

Two vectors are said to be equal if and only if (iff) they have the same magnitude and orientation.

Vector Operations

Multiple algebraic operations can be done with vectors: addition between vectors, scalar multiplication, and vector multiplication.

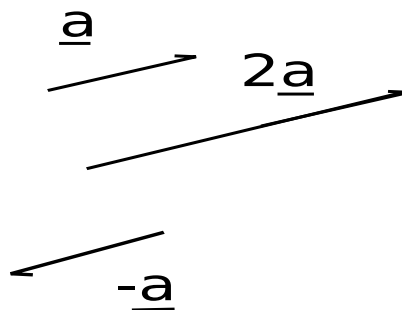
Vector addition as based on the parallelogram law.



Vector addition is also commutative:

$$\begin{aligned}\underline{a} + \underline{b} &= \underline{b} + \underline{a} \\ (\underline{a} + \underline{b}) + \underline{c} &= \underline{a} + (\underline{b} + \underline{c})\end{aligned}$$

Scalar multiplication is the product of a vector and a scalar; the result is a vector with the same orientation but whose magnitude is scaled by the scalar. Note that multiplication by -1 is equivalent to rotating the vector by 180 degrees about an arbitrary axis in the plane perpendicular to the vector.



A unit vector is simply a vector whose magnitude is equal to 1. Given any vector $\mathbf{v}$ we can define a unit vector as:

$$\hat{\mathbf{n}}_{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Note that every vector can be written as the product of a scalar and unit vector.

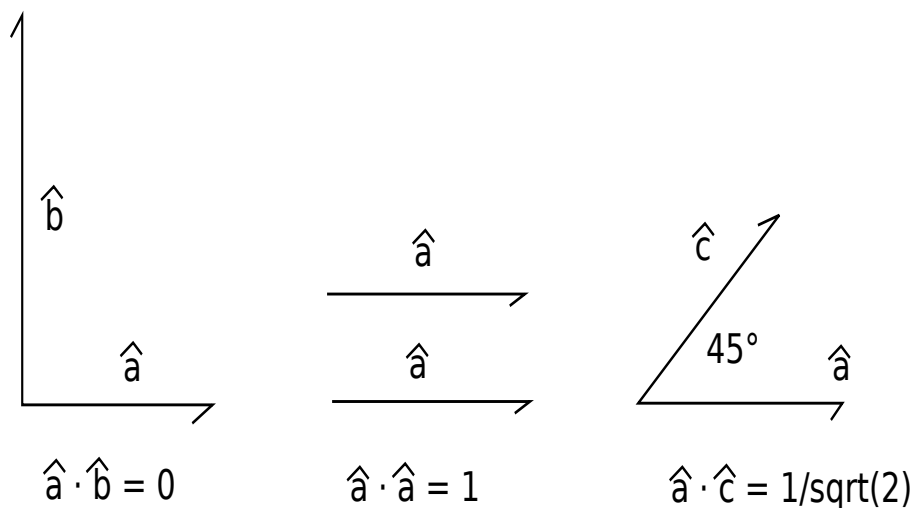
Three vector products are implemented in vector: the dot product, the cross product, and the outer product.

The dot product operation maps two vectors to a scalar. It is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta)$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

The dot product of two unit vectors represent the magnitude of the common direction; for other vectors, it is the product of the magnitude of the common direction and the two vectors' magnitudes. The dot product of two perpendicular is zero. The figure below shows some examples:



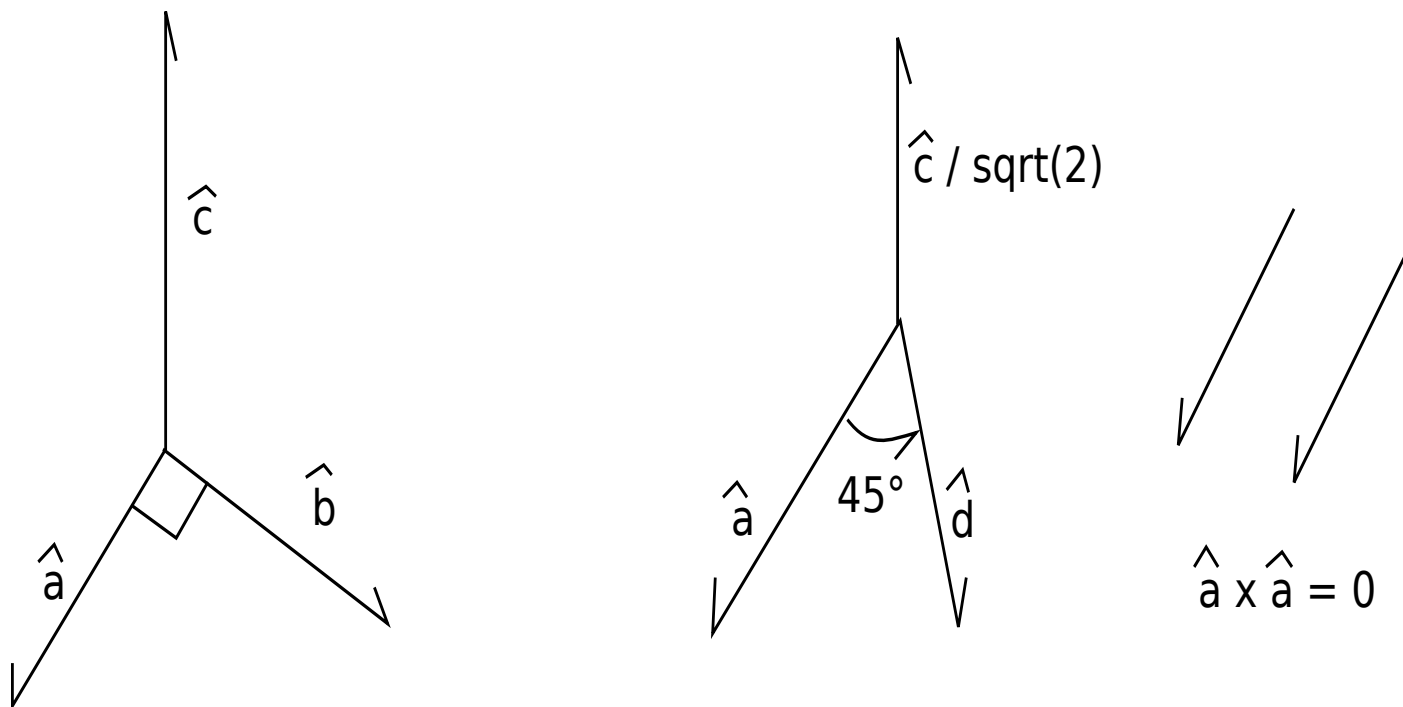
The dot product is commutative:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

The cross product vector multiplication operation of two vectors returns a vector:

$$\mathbf{a} \times \mathbf{b} = \mathbf{c}$$

The vector $\mathbf{c}$ has the following properties: it's orientation is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$, it's magnitude is defined as $\|\mathbf{c}\| = \|\mathbf{a}\| \|\mathbf{b}\| \sin(\theta)$ (where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$), and has a sense defined by using the right hand rule between $\|\mathbf{a}\| \|\mathbf{b}\|$. The figure below shows this:



The cross product has the following properties:

It is not commutative:

$$\begin{aligned}\mathbf{a} \times \mathbf{b} &\neq \mathbf{b} \times \mathbf{a} \\ \mathbf{a} \times \mathbf{b} &= -\mathbf{b} \times \mathbf{a}\end{aligned}$$

and not associative:

$$(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} \neq \mathbf{a} \times (\mathbf{b} \times \mathbf{c})$$

Two parallel vectors will have a zero cross product.

The outer product between two vectors will not be discussed here, but instead in the inertia section (that is where it is used). Other useful vector properties and relationships are:

$$\begin{aligned}\alpha(\mathbf{a} + \mathbf{b}) &= \alpha\mathbf{a} + \alpha\mathbf{b} \\ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \\ \mathbf{a} \times (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c} \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &\text{ gives the scalar triple product.} \\ \mathbf{a} \times (\mathbf{b} \cdot \mathbf{c}) &\text{ does not work, as you cannot cross a vector and a scalar.} \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= (\mathbf{b} \times \mathbf{c}) \cdot \mathbf{a} = (\mathbf{c} \times \mathbf{a}) \cdot \mathbf{b} \\ (\mathbf{a} \times \mathbf{b}) \times \mathbf{c} &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{a}(\mathbf{b} \cdot \mathbf{c}) \\ \mathbf{a} \times (\mathbf{b} \times \mathbf{c}) &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b})\end{aligned}$$

Alternative Representation

If we have three non-coplanar unit vectors $\hat{\mathbf{n}}_x, \hat{\mathbf{n}}_y, \hat{\mathbf{n}}_z$, we can represent any vector $\mathbf{a}$ as $\mathbf{a} = a_x \hat{\mathbf{n}}_x + a_y \hat{\mathbf{n}}_y + a_z \hat{\mathbf{n}}_z$. In this situation $\hat{\mathbf{n}}_x, \hat{\mathbf{n}}_y, \hat{\mathbf{n}}_z$ are referred to as a basis. a_x, a_y, a_z are called the

measure numbers. Usually the unit vectors are mutually perpendicular, in which case we can refer to them as an orthonormal basis, and they are usually right-handed.

To test equality between two vectors, now we can do the following. With vectors:

$$\begin{aligned}\mathbf{a} &= a_x \hat{\mathbf{n}}_x + a_y \hat{\mathbf{n}}_y + a_z \hat{\mathbf{n}}_z \\ \mathbf{b} &= b_x \hat{\mathbf{n}}_x + b_y \hat{\mathbf{n}}_y + b_z \hat{\mathbf{n}}_z\end{aligned}$$

We can claim equality if: $a_x = b_x, a_y = b_y, a_z = b_z$.

Vector addition is then represented, for the same two vectors, as:

$$\mathbf{a} + \mathbf{b} = (a_x + b_x) \hat{\mathbf{n}}_x + (a_y + b_y) \hat{\mathbf{n}}_y + (a_z + b_z) \hat{\mathbf{n}}_z$$

Multiplication operations are now defined as:

$$\begin{aligned}\alpha \mathbf{b} &= \alpha b_x \hat{\mathbf{n}}_x + \alpha b_y \hat{\mathbf{n}}_y + \alpha b_z \hat{\mathbf{n}}_z \\ \mathbf{a} \cdot \mathbf{b} &= a_x b_x + a_y b_y + a_z b_z \\ \mathbf{a} \times \mathbf{b} &= \det \begin{bmatrix} \hat{\mathbf{n}}_x & \hat{\mathbf{n}}_y & \hat{\mathbf{n}}_z \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} \\ (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= \det \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}\end{aligned}$$

To write a vector in a given basis, we can do the follow:

$$\mathbf{a} = (\mathbf{a} \cdot \hat{\mathbf{n}}_x) \hat{\mathbf{n}}_x + (\mathbf{a} \cdot \hat{\mathbf{n}}_y) \hat{\mathbf{n}}_y + (\mathbf{a} \cdot \hat{\mathbf{n}}_z) \hat{\mathbf{n}}_z$$

Examples

Some numeric examples of these operations follow:

$$\begin{aligned}\mathbf{a} &= \hat{\mathbf{n}}_x + 5\hat{\mathbf{n}}_y \\ \mathbf{b} &= \hat{\mathbf{n}}_y + \alpha\hat{\mathbf{n}}_z \\ \mathbf{a} + \mathbf{b} &= \hat{\mathbf{n}}_x + 6\hat{\mathbf{n}}_y + \alpha\hat{\mathbf{n}}_z \\ \mathbf{a} \cdot \mathbf{b} &= 5 \\ \mathbf{a} \cdot \hat{\mathbf{n}}_y &= 5 \\ \mathbf{a} \cdot \hat{\mathbf{n}}_z &= 0 \\ \mathbf{a} \times \mathbf{b} &= 5\alpha\hat{\mathbf{n}}_x - \alpha\hat{\mathbf{n}}_y + \hat{\mathbf{n}}_z \\ \mathbf{b} \times \mathbf{a} &= -5\alpha\hat{\mathbf{n}}_x + \alpha\hat{\mathbf{n}}_y - \hat{\mathbf{n}}_z\end{aligned}$$

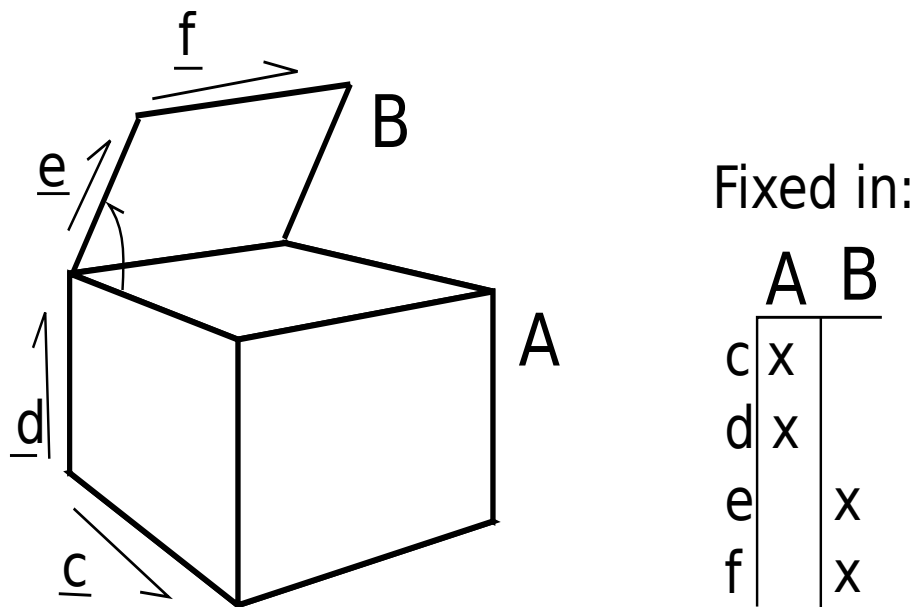
Vector Calculus

To deal with the calculus of vectors with moving object, we have to introduce the concept of a reference frame. A classic example is a train moving along its tracks, with you and a friend inside. If both you and your friend are sitting, the relative velocity between the two of you is zero. From an observer outside the train, you will both have velocity though.

We will now apply more rigor to this definition. A reference frame is a virtual “platform” which we choose to observe vector quantities from. If we have a reference frame $\mathbf{N}$, vector $\mathbf{a}$ is said to be fixed in the frame $\mathbf{N}$ if none of its properties ever change when observed from $\mathbf{N}$. We will typically assign a fixed orthonormal basis vector set with each reference frame; $\mathbf{N}$ will have $\hat{\mathbf{n}}_x, \hat{\mathbf{n}}_y, \hat{\mathbf{n}}_z$ as its basis vectors.

Derivatives of Vectors

A vector which is not fixed in a reference frame therefore has changing properties when observed from that frame. Calculus is the study of change, and in order to deal with the peculiarities of vectors fixed and not fixed in different reference frames, we need to be more explicit in our definitions.



In the above figure, we have vectors $\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}$. If one were to take the derivative of $\mathbf{e}$ with respect to θ :

$$\frac{d\mathbf{e}}{d\theta}$$

it is not clear what the derivative is. If you are observing from frame **A**, it is clearly non-zero. If you are observing from frame **B**, the derivative is zero. We will therefore introduce the frame as part of the derivative notation:

$\frac{\mathbf{A}d\mathbf{e}}{d\theta} \neq 0$, the derivative of $\mathbf{e}$ with respect to θ in the reference frame **A**

$\frac{\mathbf{B}d\mathbf{e}}{d\theta} = 0$, the derivative of $\mathbf{e}$ with respect to θ in the reference frame **B**

$\frac{\mathbf{A}d\mathbf{c}}{d\theta} = 0$, the derivative of $\mathbf{c}$ with respect to θ in the reference frame **A**

$\frac{\mathbf{B}d\mathbf{c}}{d\theta} \neq 0$, the derivative of $\mathbf{c}$ with respect to θ in the reference frame **B**

Here are some additional properties of derivatives of vectors in specific frames:

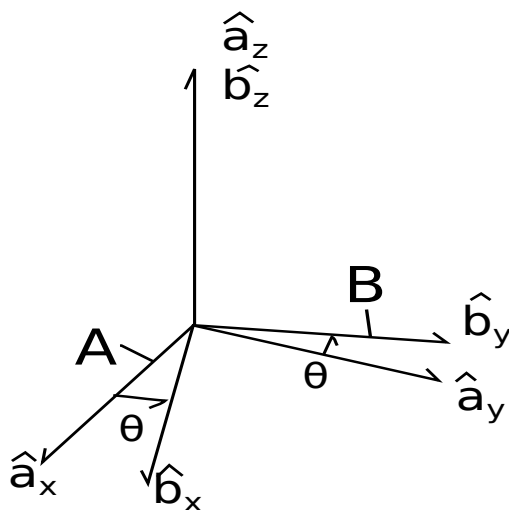
$$\begin{aligned} \frac{\mathbf{A}d}{dt}(\mathbf{a} + \mathbf{b}) &= \frac{\mathbf{A}d\mathbf{a}}{dt} + \frac{\mathbf{A}d\mathbf{b}}{dt} \\ \frac{\mathbf{A}d}{dt}\gamma\mathbf{a} &= \frac{d\gamma}{dt}\mathbf{a} + \gamma\frac{\mathbf{A}d\mathbf{a}}{dt} \\ \frac{\mathbf{A}d}{dt}(\mathbf{a} \times \mathbf{b}) &= \frac{\mathbf{A}d\mathbf{a}}{dt} \times \mathbf{b} + \mathbf{a} \times \frac{\mathbf{A}d\mathbf{b}}{dt} \end{aligned}$$

Relating Sets of Basis Vectors

We need to now define the relationship between two different reference frames; or how to relate the basis vectors of one frame to another. We can do this using a direction cosine matrix (DCM). The direction cosine matrix relates the basis vectors of one frame to another, in the following fashion:

$$\begin{bmatrix} \hat{\mathbf{a}}_x \\ \hat{\mathbf{a}}_y \\ \hat{\mathbf{a}}_z \end{bmatrix} = [\mathbf{A}\mathbf{C}\mathbf{B}] \begin{bmatrix} \hat{\mathbf{b}}_x \\ \hat{\mathbf{b}}_y \\ \hat{\mathbf{b}}_z \end{bmatrix}$$

When two frames (say, **A** & **B**) are initially aligned, then one frame has all of its basis vectors rotated around an axis which is aligned with a basis vector, we say the frames are related by a simple rotation. The figure below shows this:



The above rotation is a simple rotation about the Z axis by an angle θ . Note that after the rotation, the basis vectors $\hat{\mathbf{a}}_z$ and $\hat{\mathbf{b}}_z$ are still aligned.

This rotation can be characterized by the following direction cosine matrix:

$$\mathbf{A}\mathbf{C}\mathbf{B} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Simple rotations about the X and Y axes are defined by:

$$\text{DCM for x-axis rotation: } \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$\text{DCM for y-axis rotation: } \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Rotation in the positive direction here will be defined by using the right-hand rule.

The direction cosine matrix is also involved with the definition of the dot product between sets of basis vectors. If we have two reference frames with associated basis vectors, their direction cosine matrix can be defined as:

$$\begin{bmatrix} C_{xx} & C_{xy} & C_{xz} \\ C_{yx} & C_{yy} & C_{yz} \\ C_{zx} & C_{zy} & C_{zz} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_y & \hat{\mathbf{a}}_x \cdot \hat{\mathbf{b}}_z \\ \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_y & \hat{\mathbf{a}}_y \cdot \hat{\mathbf{b}}_z \\ \hat{\mathbf{a}}_z \cdot \hat{\mathbf{b}}_x & \hat{\mathbf{a}}_z \cdot \hat{\mathbf{b}}_y & \hat{\mathbf{a}}_z \cdot \hat{\mathbf{b}}_z \end{bmatrix}$$

Additionally, the direction cosine matrix is orthogonal, in that:

$$\begin{aligned}\mathbf{A}\mathbf{C}\mathbf{B} &= (\mathbf{B}\mathbf{C}\mathbf{A})^{-1} \\ &= (\mathbf{B}\mathbf{C}\mathbf{A})^T\end{aligned}$$

If we have reference frames **A** and **B**, which in this example have undergone a simple z-axis rotation by an amount θ , we will have two sets of basis vectors. We can then define two vectors: $\mathbf{a} = \hat{\mathbf{a}}_x + \hat{\mathbf{a}}_y + \hat{\mathbf{a}}_z$ and $\mathbf{b} = \hat{\mathbf{b}}_x + \hat{\mathbf{b}}_y + \hat{\mathbf{b}}_z$. If we wish to express **b** in the **A** frame, we do the following:

$$\begin{aligned}\mathbf{b} &= \hat{\mathbf{b}}_x + \hat{\mathbf{b}}_y + \hat{\mathbf{b}}_z \\ \mathbf{b} &= [\hat{\mathbf{a}}_x \cdot (\hat{\mathbf{b}}_x + \hat{\mathbf{b}}_y + \hat{\mathbf{b}}_z)] \hat{\mathbf{a}}_x + [\hat{\mathbf{a}}_y \cdot (\hat{\mathbf{b}}_x + \hat{\mathbf{b}}_y + \hat{\mathbf{b}}_z)] \hat{\mathbf{a}}_y + [\hat{\mathbf{a}}_z \cdot (\hat{\mathbf{b}}_x + \hat{\mathbf{b}}_y + \hat{\mathbf{b}}_z)] \hat{\mathbf{a}}_z \\ \mathbf{b} &= (\cos(\theta) - \sin(\theta))\hat{\mathbf{a}}_x + (\sin(\theta) + \cos(\theta))\hat{\mathbf{a}}_y + \hat{\mathbf{a}}_z\end{aligned}$$

And if we wish to express **a** in the **B**, we do:

$$\begin{aligned}\mathbf{a} &= \hat{\mathbf{a}}_x + \hat{\mathbf{a}}_y + \hat{\mathbf{a}}_z \\ \mathbf{a} &= [\hat{\mathbf{b}}_x \cdot (\hat{\mathbf{a}}_x + \hat{\mathbf{a}}_y + \hat{\mathbf{a}}_z)] \hat{\mathbf{b}}_x + [\hat{\mathbf{b}}_y \cdot (\hat{\mathbf{a}}_x + \hat{\mathbf{a}}_y + \hat{\mathbf{a}}_z)] \hat{\mathbf{b}}_y + [\hat{\mathbf{b}}_z \cdot (\hat{\mathbf{a}}_x + \hat{\mathbf{a}}_y + \hat{\mathbf{a}}_z)] \hat{\mathbf{b}}_z \\ \mathbf{a} &= (\cos(\theta) + \sin(\theta))\hat{\mathbf{b}}_x + (-\sin(\theta) + \cos(\theta))\hat{\mathbf{b}}_y + \hat{\mathbf{b}}_z\end{aligned}$$

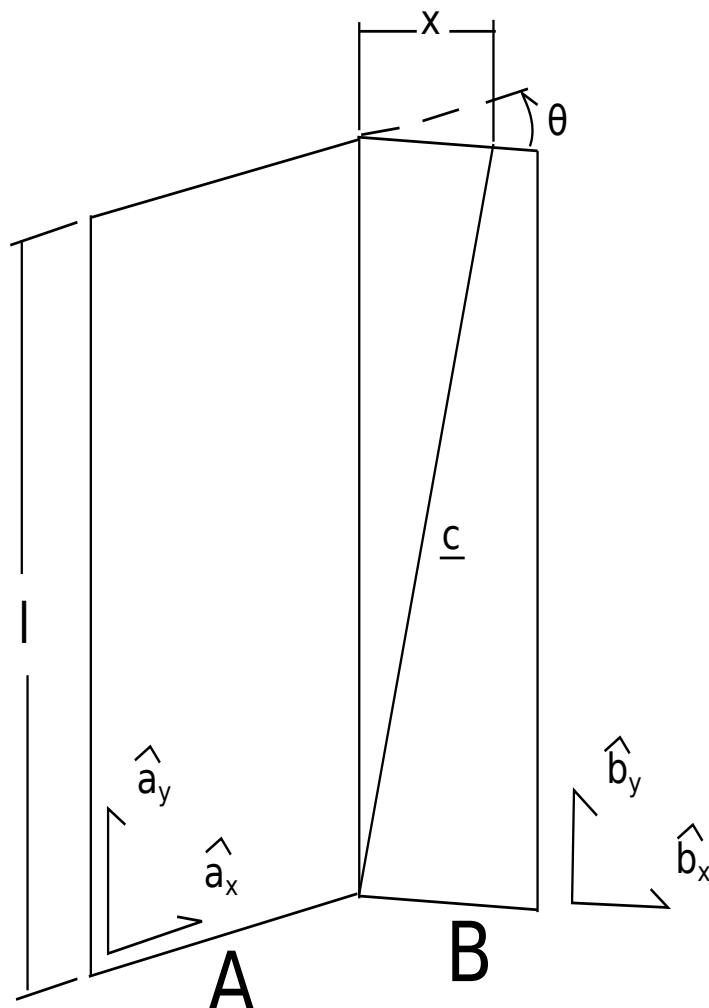
Derivatives with Multiple Frames

If we have reference frames **A** and **B** we will have two sets of basis vectors. We can then define two vectors: $\mathbf{a} = a_x\hat{\mathbf{a}}_x + a_y\hat{\mathbf{a}}_y + a_z\hat{\mathbf{a}}_z$ and $\mathbf{b} = b_x\hat{\mathbf{b}}_x + b_y\hat{\mathbf{b}}_y + b_z\hat{\mathbf{b}}_z$. If we want to take the derivative of **b** in the reference frame **A**, we must first express it in **A**, and then take the derivatives of the measure numbers:

$$\frac{\mathbf{A}d\mathbf{b}}{dx} = \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_x)}{dx}\hat{\mathbf{a}}_x + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_y)}{dx}\hat{\mathbf{a}}_y + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_z)}{dx}\hat{\mathbf{a}}_z +$$

Examples

An example of vector calculus:



In this example we have two bodies, each with an attached reference frame. We will say that θ and x are functions of time. We wish to know the time derivative of vector $\mathbf{c}$ in both the $\mathbf{A}$ and $\mathbf{B}$ frames.

First, we need to define $\mathbf{c}$; $\mathbf{c} = x\hat{\mathbf{b}}_x + l\hat{\mathbf{b}}_y$. This provides a definition in the $\mathbf{B}$ frame. We can now do the following:

$$\begin{aligned}\frac{{}^{\mathbf{B}}d\mathbf{c}}{dt} &= \frac{dx}{dt}\hat{\mathbf{b}}_x + \frac{dl}{dt}\hat{\mathbf{b}}_y \\ &= \dot{x}\hat{\mathbf{b}}_x\end{aligned}$$

To take the derivative in the $\mathbf{A}$ frame, we have to first relate the two frames:

$${}^{\mathbf{A}}\mathbf{C}^{\mathbf{B}} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Now we can do the following:

$$\begin{aligned}\frac{{}^{\mathbf{A}}d\mathbf{c}}{dt} &= \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_x)}{dt}\hat{\mathbf{a}}_x + \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_y)}{dt}\hat{\mathbf{a}}_y + \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_z)}{dt}\hat{\mathbf{a}}_z \\ &= \frac{d(\cos(\theta)x)}{dt}\hat{\mathbf{a}}_x + \frac{d(l)}{dt}\hat{\mathbf{a}}_y + \frac{d(-\sin(\theta)x)}{dt}\hat{\mathbf{a}}_z \\ &= (-\dot{\theta}\sin(\theta)x + \cos(\theta)\dot{x})\hat{\mathbf{a}}_x + (\dot{l})\hat{\mathbf{a}}_y + (\dot{\theta}\cos(\theta)x + \sin(\theta)\dot{x})\hat{\mathbf{a}}_z\end{aligned}$$

Note that this is the time derivative of $\mathbf{c}$ in $\mathbf{A}$, and is expressed in the $\mathbf{A}$ frame. We can express it in the $\mathbf{B}$ frame however, and the expression will still be valid:

$$\begin{aligned}\frac{{}^{\mathbf{A}}d\mathbf{c}}{dt} &= (-\dot{\theta} \sin(\theta)x + \cos(\theta)\dot{x})\hat{\mathbf{a}}_{\mathbf{x}} + (\dot{\theta} \cos(\theta)x + \sin(\theta)\dot{x})\hat{\mathbf{a}}_{\mathbf{z}} \\ &= \dot{x}\hat{\mathbf{b}}_{\mathbf{x}} - \theta x\hat{\mathbf{b}}_{\mathbf{z}}\end{aligned}$$

Note the difference in expression complexity between the two forms. They are equivalent, but one is much simpler. This is an extremely important concept, as defining vectors in the more complex forms can vastly slow down formulation of the equations of motion and increase their length, sometimes to a point where they cannot be shown on screen.

Using Vectors and Reference Frames

We have waited until after all of the relevant mathematical relationships have been defined for vectors and reference frames to introduce code. This is due to how vectors are formed. When starting any problem in `vector`, one of the first steps is defining a reference frame (remember to import `sympy.physics.vector` first):

```
>>> from sympy.physics.vector import *
>>> N = ReferenceFrame('N')
```

Now we have created a reference frame, $\mathbf{N}$. To have access to any basis vectors, first a reference frame needs to be created. Now that we have made an object representing $\mathbf{N}$, we can access its basis vectors:

```
>>> N.x
N.x
>>> N.y
N.y
>>> N.z
N.z
```

Vector Algebra, in `physics.vector`

We can now do basic algebraic operations on these vectors.:

```
>>> N.x == N.x
True
>>> N.x == N.y
False
>>> N.x + N.y
N.x + N.y
>>> 2 * N.x + N.y
2*N.x + N.y
```

Remember, don't add a scalar quantity to a vector (`N.x + 5`); this will raise an error. At this point, we'll use SymPy's `Symbol` in our vectors. Remember to refer to SymPy's Gotchas and Pitfalls when dealing with symbols.:

```
>>> from sympy import Symbol, symbols
>>> x = Symbol('x')
>>> x * N.x
```

(continues on next page)

(continued from previous page)

```
x*N.x
>>> x*(N.x + N.y)
x*N.x + x*N.y
```

In vector multiple interfaces to vector multiplication have been implemented, at the operator level, method level, and function level. The vector dot product can work as follows:

```
>>> N.x & N.x
1
>>> N.x & N.y
0
>>> N.x.dot(N.x)
1
>>> N.x.dot(N.y)
0
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
```

The “official” interface is the function interface; this is what will be used in all examples. This is to avoid confusion with the attribute and methods being next to each other, and in the case of the operator operation priority. The operators used in vector for vector multiplication do not possess the correct order of operations; this can lead to errors. Care with parentheses is needed when using operators to represent vector multiplication.

The cross product is the other vector multiplication which will be discussed here. It offers similar interfaces to the dot product, and comes with the same warnings.

```
>>> N.x ^ N.x
0
>>> N.x ^ N.y
N.z
>>> N.x.cross(N.x)
0
>>> N.x.cross(N.z)
- N.y
>>> cross(N.x, N.y)
N.z
>>> N.x ^ (N.y + N.z)
- N.y + N.z
```

Two additional operations can be done with vectors: normalizing the vector to length 1, and getting its magnitude. These are done as follows:

```
>>> (N.x + N.y).normalize()
sqrt(2)/2*N.x + sqrt(2)/2*N.y
>>> (N.x + N.y).magnitude()
sqrt(2)
```

Vectors are often expressed in a matrix form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame the vector is defined in, you must provide a reference frame to extract the measure numbers from the vector. There is a convenience function to do this:

```
>>> (x * N.x + 2 * x * N.y + 3 * x * N.z).to_matrix(N)
Matrix([
  [ x],
  [2*x],
  [3*x]])
```

Vector Calculus, in `physics.vector`

We have already introduced our first reference frame. We can take the derivative in that frame right now, if we desire:

```
>>> (x * N.x + N.y).diff(x, N)
N.x
```

SymPy has a `diff` function, but it does not currently work with vector `Vectors`, so please use `Vector`'s `diff` method. The reason for this is that when differentiating a `Vector`, the frame of reference must be specified in addition to what you are taking the derivative with respect to; SymPy's `diff` function doesn't fit this mold.

The more interesting case arise with multiple reference frames. If we introduce a second reference frame, **A**, we now have two frames. Note that at this point we can add components of **N** and **A** together, but cannot perform vector multiplication, as no relationship between the two frames has been defined.

```
>>> A = ReferenceFrame('A')
>>> A.x + N.x
N.x + A.x
```

If we want to do vector multiplication, first we have to define an orientation. The `orient` method of `ReferenceFrame` provides that functionality.

```
>>> A.orient(N, 'Axis', [x, N.y])
```

If we desire, we can view the DCM between these two frames at any time. This can be calculated with the `dcm` method. This code: `N.dcm(A)` gives the dcm ${}^N\mathbf{C}^A$.

This orients the **A** frame relative to the **N** frame by a simple rotation around the Y axis, by an amount x . Other, more complicated rotation types include Body rotations, Space rotations, quaternions, and arbitrary axis rotations. Body and space rotations are equivalent to doing 3 simple rotations in a row, each about a basis vector in the new frame. An example follows:

```
>>> N = ReferenceFrame('N')
>>> Bp = ReferenceFrame('Bp')
>>> Bpp = ReferenceFrame('Bpp')
>>> B = ReferenceFrame('B')
>>> q1,q2,q3 = symbols('q1 q2 q3')
>>> Bpp.orient(N, 'Axis', [q1, N.x])
>>> Bp.orient(Bpp, 'Axis', [q2, Bpp.y])
>>> B.orient(Bp, 'Axis', [q3, Bp.z])
>>> N.dcm(B)
Matrix([
  [                                cos(q2)*cos(q3),                                -
  ↪ sin(q3)*cos(q2),                                sin(q2)],
  [sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), -sin(q1)*sin(q2)*sin(q3) + ↪
  ↪ cos(q1)*cos(q3), -sin(q1)*cos(q2)],
```

(continues on next page)

(continued from previous page)

```
[sin(q1)*sin(q3) - sin(q2)*cos(q1)*cos(q3), sin(q1)*cos(q3) +
↳sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2)]]))
>>> B.orient(N, 'Body', [q1, q2, q3], 'XYZ')
>>> N.dcm(B)
Matrix([
[
cos(q2)*cos(q3),
↳sin(q3)*cos(q2), sin(q2)],
[sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), -sin(q1)*sin(q2)*sin(q3) +
↳cos(q1)*cos(q3), -sin(q1)*cos(q2)],
[sin(q1)*sin(q3) - sin(q2)*cos(q1)*cos(q3), sin(q1)*cos(q3) +
↳sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2)]]])
```

Space orientations are similar to body orientation, but applied from the frame to body. Body and space rotations can involve either two or three axes: 'XYZ' works, as does 'YZX', 'ZXZ', 'YXY', etc. What is key is that each simple rotation is about a different axis than the previous one; 'ZZX' does not completely orient a set of basis vectors in 3 space.

Sometimes it will be more convenient to create a new reference frame and orient relative to an existing one in one step. The `orientnew` method allows for this functionality, and essentially wraps the `orient` method. All of the things you can do in `orient`, you can do in `orientnew`.

```
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
```

Quaternions (or Euler Parameters) use 4 value to characterize the orientation of the frame. This and arbitrary axis rotations are described in the `orient` and `orientnew` method help, or in the references [\[Kane1983\]](#) (page 1873).

Finally, before starting multiframe calculus operations, we will introduce another vector tool: `dynamicsymbols`. `dynamicsymbols` is a shortcut function to create undefined functions of time within SymPy. The derivative of such a 'dynamicsymbol' is shown below.

```
>>> from sympy import diff
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)
```

The 'dynamicsymbol' printing is not very clear above; we will also introduce a few other tools here. We can use `vprint` instead of `print` for non-interactive sessions.

```
>>> q1
q1(t)
>>> q1d = diff(q1, Symbol('t'))
>>> vprint(q1)
q1
>>> vprint(q1d)
q1'
```

For interactive sessions use `init_vprinting`. There also exist analogs for SymPy's `vprint`, `vpprint`, and `latex`, `vlatex`.

```
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1
q1
>>> q1d
q1'
```

A ‘dynamicsymbol’ should be used to represent any time varying quantity in vector, whether it is a coordinate, varying position, or force. The primary use of a ‘dynamicsymbol’ is for speeds and coordinates (of which there will be more discussion in the Kinematics Section of the documentation).

Now we will define the orientation of our new frames with a ‘dynamicsymbol’, and can take derivatives and time derivatives with ease. Some examples follow.

```
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q1, N.x])
>>> (B.y*q2 + B.z).diff(q2, N)
B.y
>>> (B.y*q2 + B.z).dt(N)
(-q1' + q2')*B.y + q2*q1'*B.z
```

Note that the output vectors are kept in the same frames that they were provided in. This remains true for vectors with components made of basis vectors from multiple frames:

```
>>> (B.y*q2 + B.z + q2*N.x).diff(q2, N)
N.x + B.y
```

How Vectors are Coded

What follows is a short description of how vectors are defined by the code in vector. It is provided for those who want to learn more about how this part of `sympy.physics.vector` works, and does not need to be read to use this module; don’t read it unless you want to learn how this module was implemented.

Every `Vector`’s main information is stored in the `args` attribute, which stores the three measure numbers for each basis vector in a frame, for every relevant frame. A vector does not exist in code until a `ReferenceFrame` is created. At this point, the `x`, `y`, and `z` attributes of the reference frame are immutable `Vector`’s which have measure numbers of `[1,0,0]`, `[0,1,0]`, and `[0,0,1]` associated with that `ReferenceFrame`. Once these vectors are accessible, new vectors can be created by doing algebraic operations with the basis vectors. A vector can have components from multiple frames though. That is why `args` is a list; it has as many elements in the list as there are unique `ReferenceFrames` in its components, i.e. if there are A and B frame basis vectors in our new vector, `args` is of length 2; if it has A, B, and C frame basis vector, `args` is of length three.

Each element in the `args` list is a 2-tuple; the first element is a `SymPy Matrix` (this is where the measure numbers for each set of basis vectors are stored) and the second element is a `ReferenceFrame` to associate those measure numbers with.

`ReferenceFrame` stores a few things. First, it stores the name you supply it on creation (`name` attribute). It also stores the direction cosine matrices, defined upon creation with the `orientnew` method, or calling the `orient` method after creation. The direction cosine matrices are represented by `SymPy`’s `Matrix`, and are part of a dictionary where the keys are the `ReferenceFrame` and the value the `Matrix`; these are set bi-directionally; in that when you orient A to N you are setting A’s orientation dictionary to include N and its `Matrix`, but you are also setting N’s orientation dictionary to include A and its `Matrix` (that DCM being the transpose of the other).

Vector: Kinematics

This document will give some mathematical background to describing a system's kinematics as well as how to represent the kinematics in `physics.vector`.

Introduction to Kinematics

The first topic is rigid motion kinematics. A rigid body is an idealized representation of a physical object which has mass and rotational inertia. Rigid bodies are obviously not flexible. We can break down rigid body motion into translational motion, and rotational motion (when dealing with particles, we only have translational motion). Rotational motion can further be broken down into simple rotations and general rotations.

Translation of a rigid body is defined as a motion where the orientation of the body does not change during the motion; or during the motion any line segment would be parallel to itself at the start of the motion.

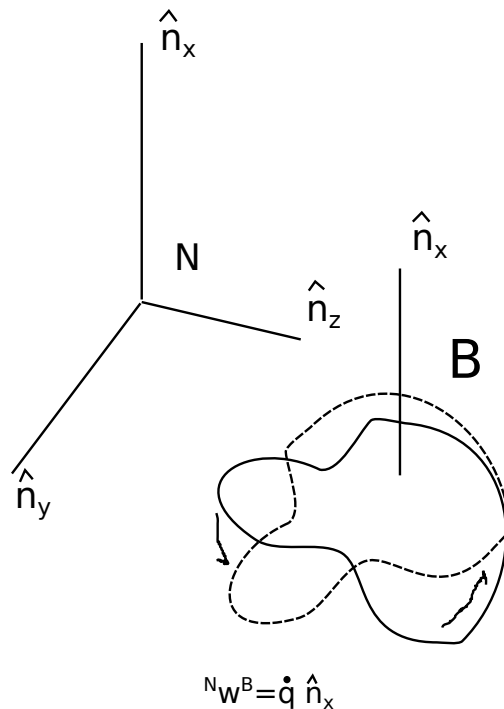
Simple rotations are rotations in which the orientation of the body may change, but there is always one line which remains parallel to itself at the start of the motion.

General rotations are rotations which there is not always one line parallel to itself at the start of the motion.

Angular Velocity

The angular velocity of a rigid body refers to the rate of change of its orientation. The angular velocity of a body is written down as: ${}^{\mathbf{N}}\omega^{\mathbf{B}}$, or the angular velocity of $\mathbf{B}$ in $\mathbf{N}$, which is a vector. Note that here, the term rigid body was used, but reference frames can also have angular velocities. Further discussion of the distinction between a rigid body and a reference frame will occur later when describing the code representation.

Angular velocity is defined as being positive in the direction which causes the orientation angles to increase (for simple rotations, or series of simple rotations).



The angular velocity vector represents the time derivative of the orientation. As a time derivative vector quantity, like those covered in the Vector & ReferenceFrame documentation, this quantity (angular velocity) needs to be defined in a reference frame. That is what the N is in the above definition of angular velocity; the frame in which the angular velocity is defined in.

The angular velocity of B in N can also be defined by:

$${}^N\omega^B = \left(\frac{{}^N d\hat{\mathbf{b}}_y}{dt} \cdot \hat{\mathbf{b}}_z\right)\hat{\mathbf{b}}_x + \left(\frac{{}^N d\hat{\mathbf{b}}_z}{dt} \cdot \hat{\mathbf{b}}_x\right)\hat{\mathbf{b}}_y + \left(\frac{{}^N d\hat{\mathbf{b}}_x}{dt} \cdot \hat{\mathbf{b}}_y\right)\hat{\mathbf{b}}_z$$

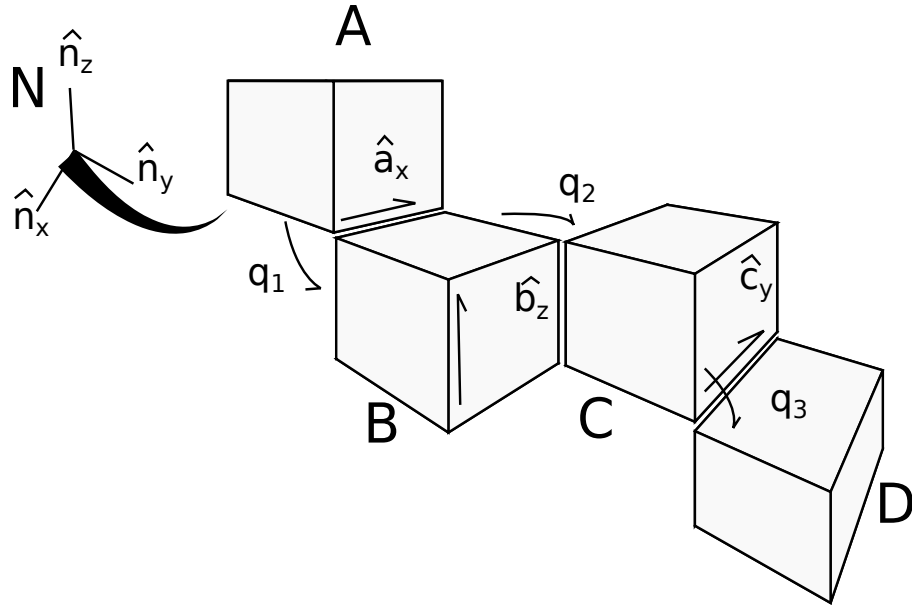
It is also common for a body's angular velocity to be written as:

$${}^N\omega^B = w_x \hat{\mathbf{b}}_x + w_y \hat{\mathbf{b}}_y + w_z \hat{\mathbf{b}}_z$$

There are a few additional important points relating to angular velocity. The first is the addition theorem for angular velocities, a way of relating the angular velocities of multiple bodies and frames. The theorem follows:

$${}^N\omega^D = {}^N\omega^A + {}^A\omega^B + {}^B\omega^C + {}^C\omega^D$$

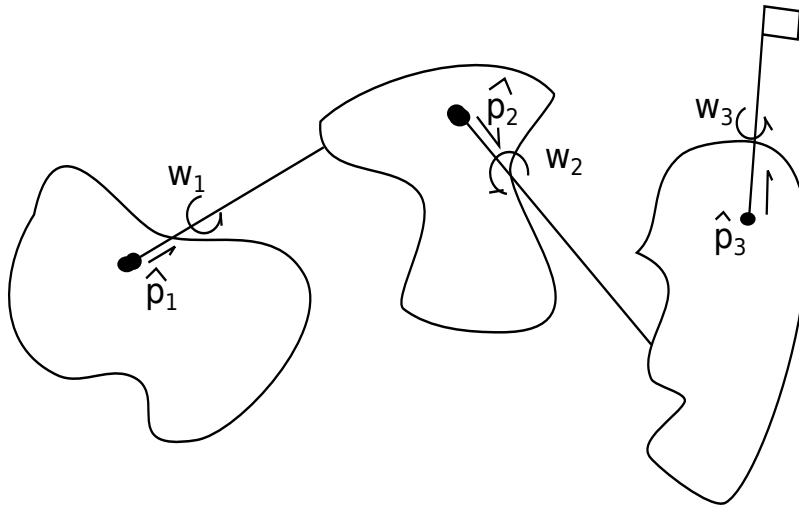
This is also shown in the following example:



$$\begin{aligned}
 {}^N\omega^A &= 0 \\
 {}^A\omega^B &= \dot{q}_1 \hat{\mathbf{a}}_x \\
 {}^B\omega^C &= -\dot{q}_2 \hat{\mathbf{b}}_z \\
 {}^C\omega^D &= \dot{q}_3 \hat{\mathbf{c}}_y \\
 {}^N\omega^D &= \dot{q}_1 \hat{\mathbf{a}}_x - \dot{q}_2 \hat{\mathbf{b}}_z + \dot{q}_3 \hat{\mathbf{c}}_y
 \end{aligned}$$

Note the signs used in the angular velocity definitions, which are related to how the displacement angle is defined in this case.

This theorem makes defining angular velocities of multibody systems much easier, as the angular velocity of a body in a chain needs to only be defined to the previous body in order to be fully defined (and the first body needs to be defined in the desired reference frame). The following figure shows an example of when using this theorem can make things easier.



Here we can easily write the angular velocity of the body **D** in the reference frame of the first body **A**:

$${}^A\omega^D = w_1 \hat{\mathbf{p}}_1 + w_2 \hat{\mathbf{p}}_2 + w_3 \hat{\mathbf{p}}_3$$

It is very important to remember to only use this with angular velocities; you cannot use this theorem with the velocities of points.

There is another theorem commonly used: the derivative theorem. It provides an alternative method (which can be easier) to calculate the time derivative of a vector in a reference frame:

$$\frac{{}^{\mathbf{N}}d\mathbf{v}}{dt} = \frac{{}^{\mathbf{B}}d\mathbf{v}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{v}$$

The vector $\mathbf{v}$ can be any vector quantity: a position vector, a velocity vector, angular velocity vector, etc. Instead of taking the time derivative of the vector in $\mathbf{N}$, we take it in $\mathbf{B}$, where $\mathbf{B}$ can be any reference frame or body, usually one in which it is easy to take the derivative on $\mathbf{v}$ in ($\mathbf{v}$ is usually composed only of the basis vector set belonging to $\mathbf{B}$). Then we add the cross product of the angular velocity of our newer frame, ${}^{\mathbf{N}}\omega^{\mathbf{B}}$ and our vector quantity $\mathbf{v}$. Again, you can choose any alternative frame for this. Examples follow:

Angular Acceleration

Angular acceleration refers to the time rate of change of the angular velocity vector. Just as the angular velocity vector is for a body and is specified in a frame, the angular acceleration vector is for a body and is specified in a frame: ${}^{\mathbf{N}}\alpha^{\mathbf{B}}$, or the angular acceleration of $\mathbf{B}$ in $\mathbf{N}$, which is a vector.

Calculating the angular acceleration is relatively straight forward:

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{N}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt}$$

Note that this can be calculated with the derivative theorem, and when the angular velocity is defined in a body fixed frame, becomes quite simple:

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{N}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt}$$

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}$$

$$\text{if } {}^{\mathbf{N}}\omega^{\mathbf{B}} = w_x \hat{\mathbf{b}}_x + w_y \hat{\mathbf{b}}_y + w_z \hat{\mathbf{b}}_z$$

$$\text{then } {}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + \underbrace{{}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}}_{\text{this is 0 by definition}}$$

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{dw_x}{dt} \hat{\mathbf{b}}_x + \frac{dw_y}{dt} \hat{\mathbf{b}}_y + \frac{dw_z}{dt} \hat{\mathbf{b}}_z$$

$${}^{\mathbf{N}}\alpha^{\mathbf{B}} = \dot{w}_x \hat{\mathbf{b}}_x + \dot{w}_y \hat{\mathbf{b}}_y + \dot{w}_z \hat{\mathbf{b}}_z$$

Again, this is only for the case in which the angular velocity of the body is defined in body fixed components.

Point Velocity & Acceleration

Consider a point, P : we can define some characteristics of the point. First, we can define a position vector from some other point to P . Second, we can define the velocity vector of P in a reference frame of our choice. Third, we can define the acceleration vector of P in a reference frame of our choice.

These three quantities are read as:

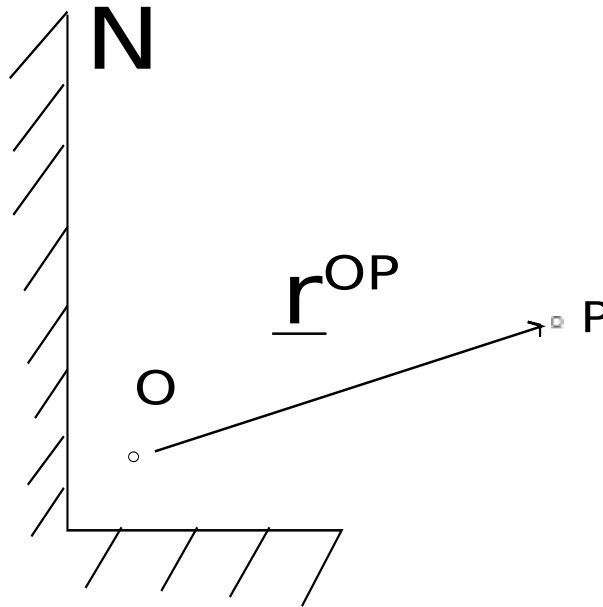
$\mathbf{r}^{OP}$, the position vector from O to P

${}^N\mathbf{v}^P$, the velocity of P in the reference frame $\mathbf{N}$

${}^N\mathbf{a}^P$, the acceleration of P in the reference frame $\mathbf{N}$

Note that the position vector does not have a frame associated with it; this is because there is no time derivative involved, unlike the velocity and acceleration vectors.

We can find these quantities for a simple example easily:



Let's define: $\mathbf{r}^{OP} = q_x \hat{\mathbf{n}}_x + q_y \hat{\mathbf{n}}_y$

$${}^N\mathbf{v}^P = \frac{{}^N d\mathbf{r}^{OP}}{dt}$$

then we can calculate: ${}^N\mathbf{v}^P = \dot{q}_x \hat{\mathbf{n}}_x + \dot{q}_y \hat{\mathbf{n}}_y$

$$\text{and } {}^N\mathbf{a}^P = \frac{{}^N d({}^N\mathbf{v}^P)}{dt}$$

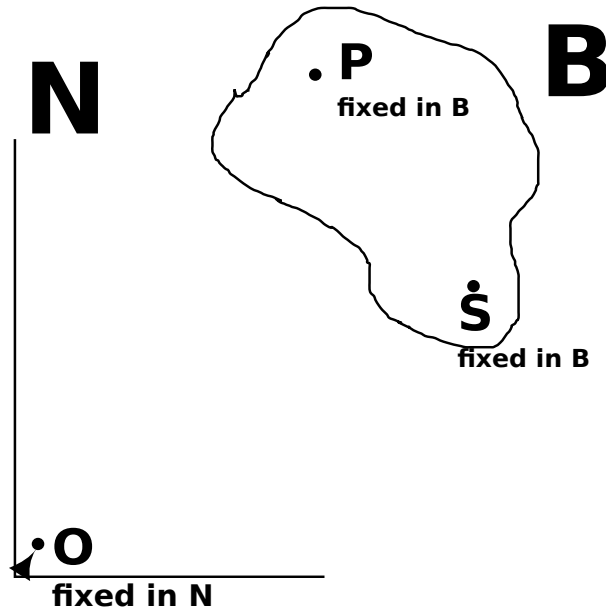
$${}^N\mathbf{a}^P = \ddot{q}_x \hat{\mathbf{n}}_x + \ddot{q}_y \hat{\mathbf{n}}_y$$

It is critical to understand in the above example that the point O is fixed in the reference frame $\mathbf{N}$. There is no addition theorem for translational velocities; alternatives will be discussed later though. Also note that the position of every point might not always need to be defined to form the dynamic equations of motion. When you don't want to define the position vector of a point, you can start by just defining the velocity vector. For the above example:

Let us instead define the velocity vector as: ${}^N\mathbf{v}^P = u_x \hat{\mathbf{n}}_x + u_y \hat{\mathbf{n}}_y$

then acceleration can be written as: ${}^N\mathbf{a}^P = \dot{u}_x \hat{\mathbf{n}}_x + \dot{u}_y \hat{\mathbf{n}}_y$

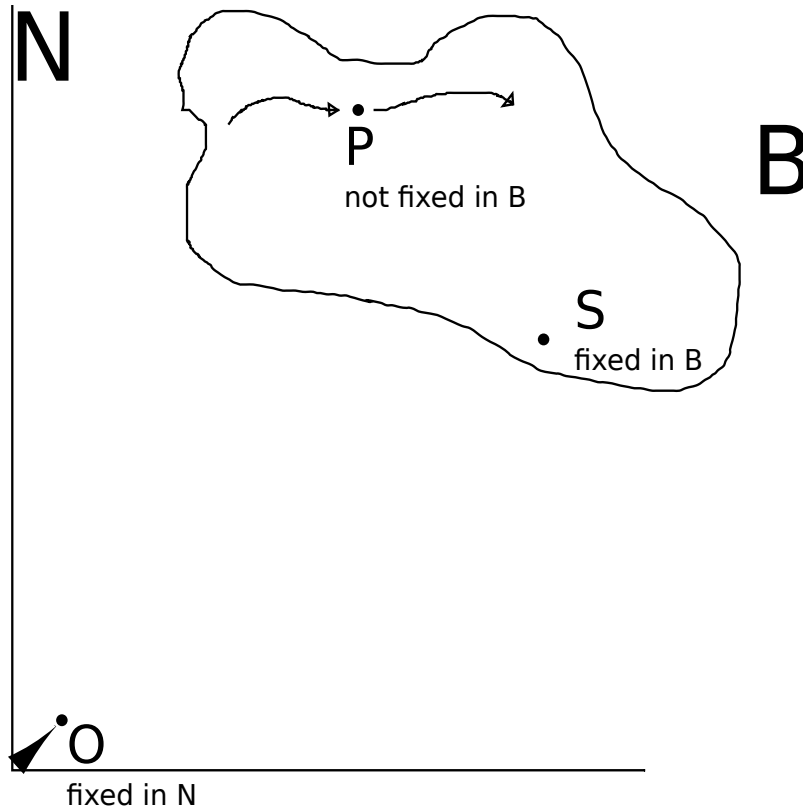
There will often be cases when the velocity of a point is desired and a related point's velocity is known. For the cases in which we have two points fixed on a rigid body, we use the 2-Point Theorem:



Let's say we know the velocity of the point S and the angular velocity of the body $\mathbf{B}$, both defined in the reference frame $\mathbf{N}$. We can calculate the velocity and acceleration of the point P in $\mathbf{N}$ as follows:

$$\begin{aligned}\mathbf{N}\mathbf{v}^P &= \mathbf{N}\mathbf{v}^S + \mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP} \\ \mathbf{N}\mathbf{a}^P &= \mathbf{N}\mathbf{a}^S + \mathbf{N}\boldsymbol{\alpha}^{\mathbf{B}} \times \mathbf{r}^{SP} + \mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times (\mathbf{N}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP})\end{aligned}$$

When only one of the two points is fixed on a body, the 1 point theorem is used instead.

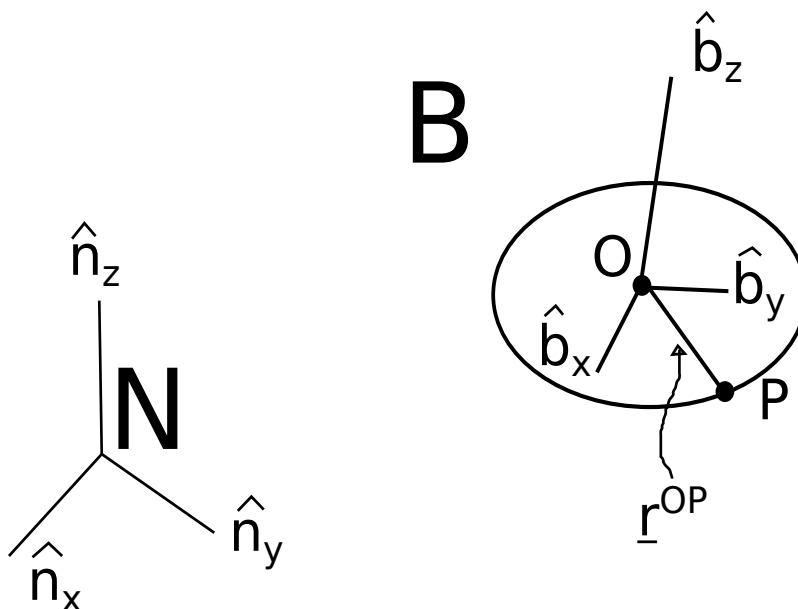


Here, the velocity of point S is known in the frame $\mathbf{N}$, the angular velocity of $\mathbf{B}$ is known in $\mathbf{N}$, and the velocity of the point P is known in the frame associated with body $\mathbf{B}$. We can then write the velocity and acceleration of P in $\mathbf{N}$ as:

$${}^{\mathbf{N}}\mathbf{v}^P = {}^{\mathbf{B}}\mathbf{v}^P + {}^{\mathbf{N}}\mathbf{v}^S + {}^{\mathbf{N}}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP}$$

$${}^{\mathbf{N}}\mathbf{a}^P = {}^{\mathbf{B}}\mathbf{a}^S + {}^{\mathbf{N}}\mathbf{a}^O + {}^{\mathbf{N}}\boldsymbol{\alpha}^{\mathbf{B}} \times \mathbf{r}^{SP} + {}^{\mathbf{N}}\boldsymbol{\omega}^{\mathbf{B}} \times ({}^{\mathbf{N}}\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{SP}) + 2{}^{\mathbf{N}}\boldsymbol{\omega}^{\mathbf{B}} \times {}^{\mathbf{B}}\mathbf{v}^P$$

Examples of applications of the 1 point and 2 point theorem follow.



This example has a disc translating and rotating in a plane. We can easily define the angular velocity of the body **B** and velocity of the point *O*:

$${}^N\omega^{\mathbf{B}} = u_3 \hat{\mathbf{n}}_z = u_3 \hat{\mathbf{b}}_z$$

$${}^N\mathbf{v}^O = u_1 \hat{\mathbf{n}}_x + u_2 \hat{\mathbf{n}}_y$$

and accelerations can be written as:

$${}^N\alpha^{\mathbf{B}} = \dot{u}_3 \hat{\mathbf{n}}_z = \dot{u}_3 \hat{\mathbf{b}}_z$$

$${}^N\mathbf{a}^O = \dot{u}_1 \hat{\mathbf{n}}_x + \dot{u}_2 \hat{\mathbf{n}}_y$$

We can use the 2 point theorem to calculate the velocity and acceleration of point *P* now.

$$\mathbf{r}^{OP} = R \hat{\mathbf{b}}_x$$

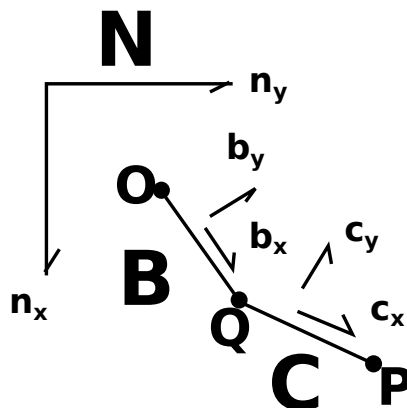
$${}^N\mathbf{v}^P = {}^N\mathbf{v}^O + {}^N\omega^{\mathbf{B}} \times \mathbf{r}^{OP}$$

$${}^N\mathbf{v}^P = u_1 \hat{\mathbf{n}}_x + u_2 \hat{\mathbf{n}}_y + u_3 \hat{\mathbf{b}}_z \times R \hat{\mathbf{b}}_x = u_1 \hat{\mathbf{n}}_x + u_2 \hat{\mathbf{n}}_y + u_3 R \hat{\mathbf{b}}_y$$

$${}^N\mathbf{a}^P = {}^N\mathbf{a}^O + {}^N\alpha^{\mathbf{B}} \times \mathbf{r}^{OP} + {}^N\omega^{\mathbf{B}} \times ({}^N\omega^{\mathbf{B}} \times \mathbf{r}^{OP})$$

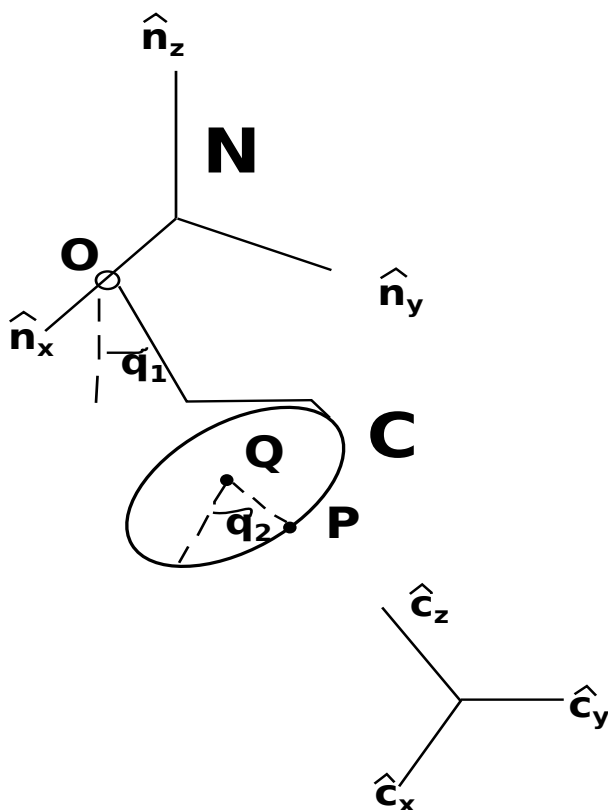
$${}^N\mathbf{a}^P = \dot{u}_1 \hat{\mathbf{n}}_x + \dot{u}_2 \hat{\mathbf{n}}_y + \dot{u}_3 \hat{\mathbf{b}}_z \times R \hat{\mathbf{b}}_x + u_3 \hat{\mathbf{b}}_z \times (u_3 \hat{\mathbf{b}}_z \times R \hat{\mathbf{b}}_x)$$

$${}^N\mathbf{a}^P = \dot{u}_1 \hat{\mathbf{n}}_x + \dot{u}_2 \hat{\mathbf{n}}_y + R \dot{u}_3 \hat{\mathbf{b}}_y - R u_3^2 \hat{\mathbf{b}}_x$$



In this example we have a double pendulum. We can use the two point theorem twice here in order to find the velocity of points Q and P ; point O 's velocity is zero in $\mathbf{N}$.

$$\begin{aligned}\mathbf{r}^{OQ} &= l\hat{\mathbf{b}}_x \\ \mathbf{r}^{QP} &= l\hat{\mathbf{c}}_x \\ {}^N\boldsymbol{\omega}^{\mathbf{B}} &= u_1\hat{\mathbf{b}}_z \\ {}^N\boldsymbol{\omega}^{\mathbf{C}} &= u_2\hat{\mathbf{c}}_z \\ {}^N\mathbf{v}^Q &= {}^N\mathbf{v}^O + {}^N\boldsymbol{\omega}^{\mathbf{B}} \times \mathbf{r}^{OQ} \\ {}^N\mathbf{v}^Q &= u_1l\hat{\mathbf{b}}_y \\ {}^N\mathbf{v}^P &= {}^N\mathbf{v}^Q + {}^N\boldsymbol{\omega}^{\mathbf{C}} \times \mathbf{r}^{QP} \\ {}^N\mathbf{v}^P &= u_1l\hat{\mathbf{b}}_y + u_2\hat{\mathbf{c}}_z \times l\hat{\mathbf{c}}_x \\ {}^N\mathbf{v}^P &= u_1l\hat{\mathbf{b}}_y + u_2l\hat{\mathbf{c}}_y\end{aligned}$$

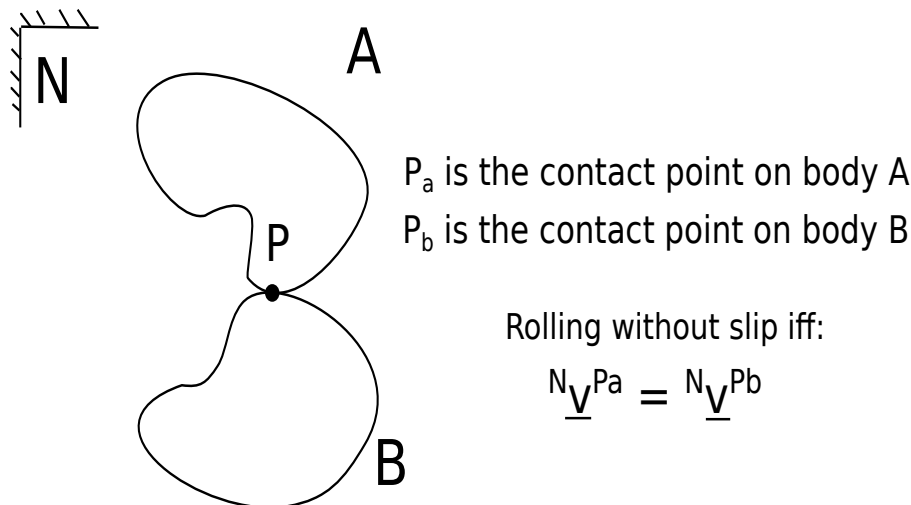


In this example we have a particle moving on a ring; the ring is supported by a rod which can rotate about the $\hat{\mathbf{n}}_x$ axis. First we use the two point theorem to find the velocity of the center point of the ring, Q , then use the 1 point theorem to find the velocity of the particle on the

ring.

$$\begin{aligned}
 {}^N\omega^C &= u_1 \hat{\mathbf{n}}_x \\
 \mathbf{r}^{OQ} &= -l \hat{\mathbf{c}}_z \\
 {}^N\mathbf{v}^Q &= u_1 l \hat{\mathbf{c}}_y \\
 \mathbf{r}^{QP} &= R(\cos(q_2) \hat{\mathbf{c}}_x + \sin(q_2) \hat{\mathbf{c}}_y) \\
 {}^C\mathbf{v}^P &= Ru_2(-\sin(q_2) \hat{\mathbf{c}}_x + \cos(q_2) \hat{\mathbf{c}}_y) \\
 {}^N\mathbf{v}^P &= {}^C\mathbf{v}^P + {}^N\mathbf{v}^Q + {}^N\omega^C \times \mathbf{r}^{QP} \\
 {}^N\mathbf{v}^P &= Ru_2(-\sin(q_2) \hat{\mathbf{c}}_x + \cos(q_2) \hat{\mathbf{c}}_y) + u_1 l \hat{\mathbf{c}}_y + u_1 \hat{\mathbf{c}}_x \times R(\cos(q_2) \hat{\mathbf{c}}_x + \sin(q_2) \hat{\mathbf{c}}_y) \\
 {}^N\mathbf{v}^P &= -Ru_2 \sin(q_2) \hat{\mathbf{c}}_x + (Ru_2 \cos(q_2) + u_1 l) \hat{\mathbf{c}}_y + Ru_1 \sin(q_2) \hat{\mathbf{c}}_z
 \end{aligned}$$

A final topic in the description of velocities of points is that of rolling, or rather, rolling without slip. Two bodies are said to be rolling without slip if and only if the point of contact on each body has the same velocity in another frame. See the following figure:



This is commonly used to form the velocity of a point on one object rolling on another fixed object, such as in the following example:

Kinematics in physics.vector

It should be clear by now that the topic of kinematics here has been mostly describing the correct way to manipulate vectors into representing the velocities of points. Within vector there are convenient methods for storing these velocities associated with frames and points. We'll now revisit the above examples and show how to represent them in [sympy](#) (page 776).

The topic of reference frame creation has already been covered. When a ReferenceFrame is created though, it automatically calculates the angular velocity of the frame using the time derivative of the DCM and the angular velocity definition.

```

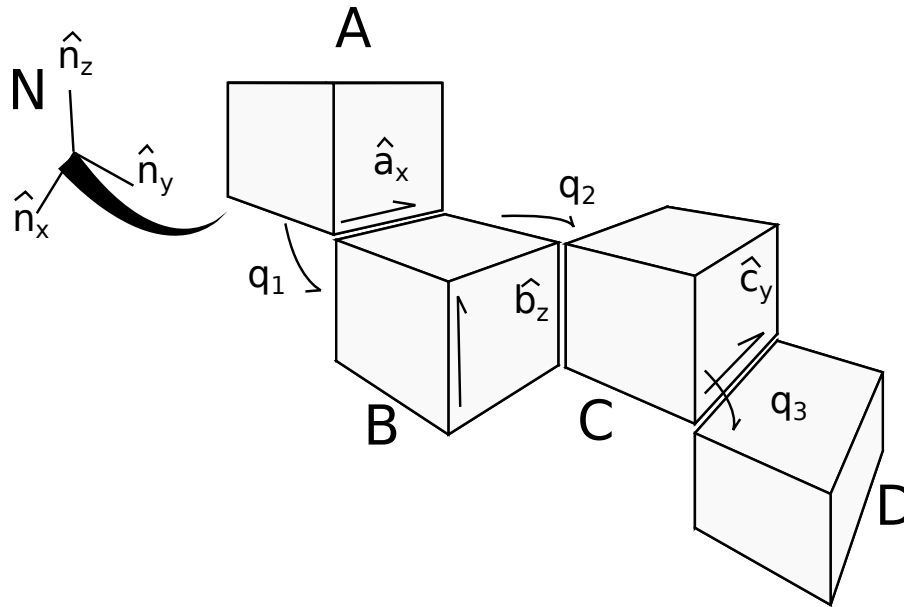
>>> from sympy import Symbol, sin, cos
>>> from sympy.physics.vector import *
>>> N = ReferenceFrame('N')
>>> q1 = dynamicsymbols('q1')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.ang_vel_in(N)
q1'*N.x

```

Note that the angular velocity can be defined in an alternate way:

```
>>> B = ReferenceFrame('B')
>>> u1 = dynamicsymbols('u1')
>>> B.set_ang_vel(N, u1 * B.y)
>>> B.ang_vel_in(N)
u1*B.y
>>> N.ang_vel_in(B)
- u1*B.y
```

Both upon frame creation during `orientnew` and when calling `set_ang_vel`, the angular velocity is set in both frames involved, as seen above.



Here we have multiple bodies with angular velocities defined relative to each other. This is coded as:

```
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> C = ReferenceFrame('C')
>>> D = ReferenceFrame('D')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> A.set_ang_vel(N, 0)
>>> B.set_ang_vel(A, u1 * A.x)
>>> C.set_ang_vel(B, -u2 * B.z)
>>> D.set_ang_vel(C, u3 * C.y)
>>> D.ang_vel_in(N)
u1*A.x - u2*B.z + u3*C.y
```

In vector the shortest path between two frames is used when finding the angular velocity. That would mean if we went back and set:

```
>>> D.set_ang_vel(N, 0)
>>> D.ang_vel_in(N)
0
```

The path that was just defined is what is used. This can cause problems though, as now the angular velocity definitions are inconsistent. It is recommended that you avoid doing this.

Points are a translational analog to the rotational ReferenceFrame. Creating a Point can be done in two ways, like ReferenceFrame:

```
>>> O = Point('O')
>>> P = O.locatenew('P', 3 * N.x + N.y)
>>> P.pos_from(O)
3*N.x + N.y
>>> Q = Point('Q')
>>> Q.set_pos(P, N.z)
>>> Q.pos_from(P)
N.z
>>> Q.pos_from(O)
3*N.x + N.y + N.z
```

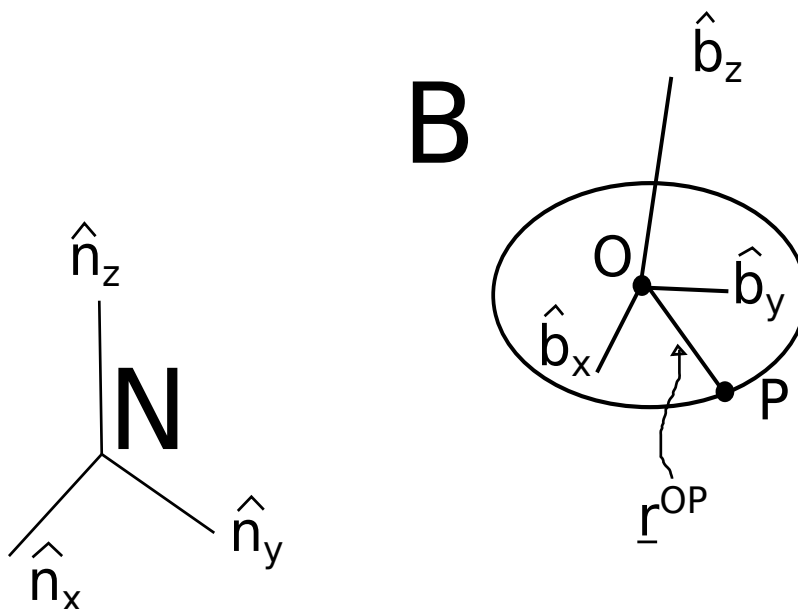
Similar to ReferenceFrame, the position vector between two points is found by the shortest path (number of intermediate points) between them. Unlike rotational motion, there is no addition theorem for the velocity of points. In order to have the velocity of a Point in a ReferenceFrame, you have to set the value.

```
>>> O = Point('O')
>>> O.set_vel(N, u1*N.x)
>>> O.vel(N)
u1*N.x
```

For both translational and rotational accelerations, the value is computed by taking the time derivative of the appropriate velocity, unless the user sets it otherwise.

```
>>> O.acc(N)
u1'*N.x
>>> O.set_acc(N, u2*u1*N.y)
>>> O.acc(N)
u1*u2*N.y
```

Next is a description of the 2 point and 1 point theorems, as used in sympy.



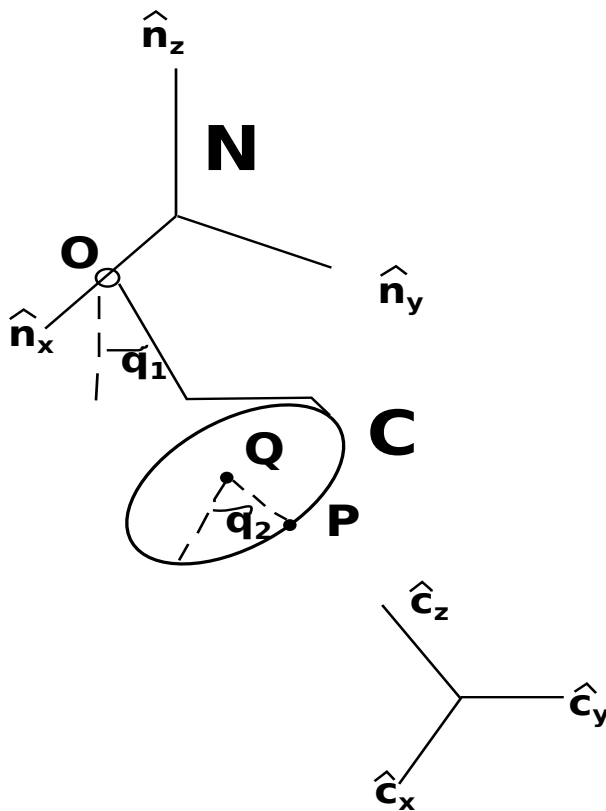
First is the translating, rotating disc.

```

>>> N = ReferenceFrame('N')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> R = Symbol('R')
>>> B = ReferenceFrame('B')
>>> O = Point('O')
>>> O.set_vel(N, u1 * N.x + u2 * N.y)
>>> P = O.locatenew('P', R * B.x)
>>> B.set_ang_vel(N, u3 * B.z)
>>> P.v2pt_theory(O, N, B)
u1*N.x + u2*N.y + R*u3*B.y
>>> P.a2pt_theory(O, N, B)
u1'*N.x + u2'*N.y - R*u3**2*B.x + R*u3'*B.y

```

We will also cover implementation of the 1 point theorem.



This is the particle moving on a ring, again.

```

>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> l = Symbol('l')
>>> R = Symbol('R')
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
>>> C.set_ang_vel(N, u1 * N.x)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> Q = O.locatenew('Q', -l * C.z)
>>> P = Q.locatenew('P', R * (cos(q2) * C.x + sin(q2) * C.y))
>>> P.set_vel(C, R * u2 * (-sin(q2) * C.x + cos(q2) * C.y))
>>> Q.v2pt_theory(O, N, C)

```

(continues on next page)

(continued from previous page)

```
l*u1*C.y
>>> P.vlpt_theory(Q, N, C)
- R*u2*sin(q2)*C.x + (R*u2*cos(q2) + l*u1)*C.y + R*u1*sin(q2)*C.z
```

Potential Issues/Advanced Topics/Future Features in Physics/Vector Module

This document will describe some of the more advanced functionality that this module offers but which is not part of the “official” interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.

Inertia (Dyadics)

A dyadic tensor is a second order tensor formed by the juxtaposition of a pair of vectors. There are various operations defined with respect to dyadics, which have been implemented in vector in the form of class `Dyadic`. To know more, refer to the `Dyadic` and `Vector` class APIs. Dyadics are used to define the inertia of bodies within mechanics. Inertia dyadics can be defined explicitly but the `inertia` function is typically much more convenient for the user:

```
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
```

Supply a reference frame and the moments of inertia if the object is symmetrical:

```
>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
```

Supply a reference frame along with the products and moments of inertia for a general object:

```
>>> inertia(N, 1, 2, 3, 4, 5, 6)
(N.x|N.x) + 4*(N.x|N.y) + 6*(N.x|N.z) + 4*(N.y|N.x) + 2*(N.y|N.y) + 5*(N.y|N.z) +
↪ 6*(N.z|N.x) + 5*(N.z|N.y) + 3*(N.z|N.z)
```

Notice that the `inertia` function returns a dyadic with each component represented as two unit vectors separated by a `|`. Refer to the [Dyadic](#) (page 1585) section for more information about dyadics.

Inertia is often expressed in a matrix, or tensor, form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame(s) the inertia dyadic is defined in, you must provide one or two reference frames to extract the measure numbers from the dyadic. There is a convenience function to do this:

```
>>> inertia(N, 1, 2, 3, 4, 5, 6).to_matrix(N)
Matrix([
[1, 4, 6],
[4, 2, 5],
[6, 5, 3]])
```

Common Issues

Here issues with numerically integrating code, choice of *dynamicsymbols* for coordinate and speed representation, printing, differentiating, and substitution will occur.

Printing

The default printing options are to use sorting for Vector and Dyadic measure numbers, and have unsorted output from the `vprint`, `vpprint`, and `vlatex` functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

Substitution

Substitution into large expressions can be slow, and take a few minutes.

Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorted expressions at this point, which will result in shorter expressions later (such as when forming Kane's equations).

Advanced Interfaces

Here we will cover advanced options in: `ReferenceFrame`, `dynamicsymbols`, and some associated functionality.

ReferenceFrame

`ReferenceFrame` is shown as having a `.name` attribute and `.x`, `.y`, and `.z` attributes for accessing the basis vectors, as well as a fairly rigidly defined print output. If you wish to have a different set of indices defined, there is an option for this. This will also require a different interface for accessing the basis vectors.

```
>>> from sympy.physics.vector import ReferenceFrame, vprint, vpprint, vlatex
>>> N = ReferenceFrame('N', indices=['i', 'j', 'k'])
>>> N['i']
N['i']
>>> N.x
N['i']
>>> vlatex(N.x)
'\mathbf{\hat{n}_{i}}'
```

Also, the latex output can have custom strings; rather than just indices though, the entirety of each basis vector can be specified. The custom latex strings can occur without custom indices, and also overwrites the latex string that would be used if there were custom indices.

```
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> N = ReferenceFrame('N', latexs=['n1', '\mathbf{n}_2', 'cat'])
>>> vlatex(N.x)
'n1'
>>> vlatex(N.y)
'\mathbf{n}_2'
>>> vlatex(N.z)
'cat'
```

dynamicsymbols

The `dynamicsymbols` function also has ‘hidden’ functionality; the variable which is associated with time can be changed, as well as the notation for printing derivatives.

```
>>> from sympy import symbols
>>> from sympy.physics.vector import dynamicsymbols, vprint
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> dynamicsymbols._t = symbols('T')
>>> q2 = dynamicsymbols('q2')
>>> q2
q2(T)
>>> q1
q1(t)
>>> q1d = dynamicsymbols('q1', 1)
>>> vprint(q1d)
q1'
>>> dynamicsymbols._str = 'd'
>>> vprint(q1d)
q1d
>>> dynamicsymbols._str = '\''
>>> dynamicsymbols._t = symbols('t')
```

Note that only dynamic symbols created after the change are different. The same is not true for the `.str` attribute; this affects the printing output only, so dynamic symbols created before or after will print the same way.

Also note that `Vector`’s `.dt` method uses the `._t` attribute of `dynamicsymbols`, along with a number of other important functions and methods. Don’t mix and match symbols representing time.

Scalar and Vector Field Functionality

Introduction

Vectors and Scalars

In physics, we deal with two kinds of quantities – scalars and vectors.

A scalar is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A vector, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector $\mathbf{V}$, the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\hat{\mathbf{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}})$ m, where, as per standard convention, $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ represent unit vectors in the $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ directions respectively. Therefore, it can be concluded that the distance traveled is $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\| \text{ m} = 5\sqrt{2} \text{ m}$. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$.

Fields

In general, a *field* is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). In this module, we deal with 3-dimensional spaces only. Hence, a field is defined as a function of the x , y and z coordinates corresponding to a location in 3D space.

For example, temperature in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location (x, y, z) of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $g(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\hat{\mathbf{i}} + 2x^2\hat{\mathbf{j}}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Implementation of fields in sympy.physics.vector

In `sympy.physics.vector`, every `ReferenceFrame` instance is assigned basis vectors corresponding to the X , Y and Z directions. These can be accessed using the attributes named `x`, `y` and `z` respectively. Hence, to define a vector $\mathbf{v}$ of the form $3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}$ with respect to a given frame $\mathbf{R}$, you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> v = 3*R.x + 4*R.y + 5*R.z
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in other sections of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented as special SymPy Symbols assigned to every frame, one for each direction from X , Y and Z . For a frame $\mathbf{R}$, the X , Y and Z base scalar Symbols can be accessed using the `R[0]`, `R[1]` and `R[2]` expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> electric_potential
2*R_x**2*R_y
```

In string representation, R_x denotes the X base scalar assigned to ReferenceFrame R . Essentially, R_x is the string representation of $R[0]$.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to x (i.e. $R[0]$), you would have to use the `diff` method.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy import diff
>>> diff(electric_potential, R[0])
4*R_x*R_y
```

Like vectors (and vector fields), scalar fields can also be re-expressed in other frames of reference, apart from the one they were defined in – assuming that an orientation relationship exists between the concerned frames. This can be done using the `express` method, in a way similar to vectors – but with the `variables` parameter set to `True`.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy.physics.vector import dynamicsymbols, express
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> express(electric_potential, R1, variables=True)
2*(R1_x*sin(q(t)) + R1_y*cos(q(t)))*(R1_x*cos(q(t)) - R1_y*sin(q(t)))**2
```

Moreover, considering scalars can also be functions of time just as vectors, differentiation with respect to time is also possible. Depending on the Symbols present in the expression and the frame with respect to which the time differentiation is being done, the output will change/remain the same.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(electric_potential, R)
0
>>> time_derivative(electric_potential, R1).simplify()
2*(R1_x*cos(q(t)) - R1_y*sin(q(t)))*(3*R1_x**2*cos(2*q(t))/2 -
R1_x**2/2 - 3*R1_x*R1_y*sin(2*q(t)) - 3*R1_y**2*cos(2*q(t))/2 -
R1_y**2/2)*Derivative(q(t), t)
```

Field operators and other related functions

Here we describe some basic field-related functionality implemented in `sympy.physics.vector`

Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector $\mathbf{F}$, denoted by $\nabla \times \mathbf{F}$ is given by -

$$\nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{\mathbf{i}} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{\mathbf{j}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{\mathbf{k}}$$

where F_x denotes the X component of vector $\mathbf{F}$.

To compute the curl of a vector field in `physics.vector`, you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import curl
>>> field = R[0]*R[1]*R[2]*R.x
>>> curl(field, R)
R_x*R_y*R.y - R_x*R_z*R.z
```

Divergence

Divergence is a vector operator that measures the magnitude of a vector field's source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector $\mathbf{F}$, denoted by $\nabla \cdot \mathbf{F}$ is given by -

$$\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}$$

where U , V and W denote the X , Y and Z components of $\mathbf{F}$ respectively.

To compute the divergence of a vector field in `physics.vector`, you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import divergence
>>> field = R[0]*R[1]*R[2] * (R.x+R.y+R.z)
>>> divergence(field, R)
R_x*R_y + R_x*R_z + R_y*R_z
```

Gradient

Consider a scalar field $f(x, y, z)$ in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of f with respect to x , y and z in the X , Y and Z directions respectively.

In the 3D Cartesian system, the gradient of a scalar field f , denoted by ∇f is given by -

$$\nabla f = \frac{\partial f}{\partial x} \hat{\mathbf{i}} + \frac{\partial f}{\partial y} \hat{\mathbf{j}} + \frac{\partial f}{\partial z} \hat{\mathbf{k}}$$

To compute the gradient of a scalar field in `physics.vector`, you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import gradient
>>> scalar_field = R[0]*R[1]*R[2]
```

(continues on next page)

(continued from previous page)

```
>>> gradient(scalar_field, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_x*R_y*R.z
```

Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path between them. A conservative vector field is also said to be ‘irrotational’, since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in `physics.vector`, use the `is_conservative` function.

```
>>> from sympy.physics.vector import ReferenceFrame, is_conservative
>>> R = ReferenceFrame('R')
>>> field = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> is_conservative(field)
True
>>> curl(field, R)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in `physics.vector`, use the `is_solenoidal` function.

```
>>> from sympy.physics.vector import ReferenceFrame, is_solenoidal
>>> R = ReferenceFrame('R')
>>> field = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> is_solenoidal(field)
True
>>> divergence(field, R)
0
```

Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the ‘scalar potential field’ corresponding to the aforementioned conservative field.

The `scalar_potential` function in `physics.vector` calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```
>>> from sympy.physics.vector import ReferenceFrame, scalar_potential
>>> R = ReferenceFrame('R')
>>> conservative_field = 4*R[0]*R[1]*R[2]*R.x + 2*R[0]**2*R[2]*R.y + 2*R[0]**2*R[1]*R.
↪ z
>>> scalar_potential(conservative_field, R)
2*R_x**2*R_y*R_z
```

Providing a non-conservative vector field as an argument to `scalar_potential` raises a `ValueError`.

The scalar potential difference, or simply ‘potential difference’, corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in `physics.vector`.

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', 1*R.x + 2*R.y + 3*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
4
```

If provided with a scalar expression instead of a vector field, `scalar_potential_difference` returns the difference between the values of that scalar field at the two given points in space.

Vector API

Essential Classes

CoordinateSym

class `sympy.physics.vector.frame.CoordinateSym`

A coordinate symbol/base scalar associated wrt a Reference Frame.

Ideally, users should not instantiate this class. Instances of this class must only be accessed through the corresponding frame as `frame[index]`.

CoordinateSyms having the same frame and index parameters are equal (even though they may be instantiated separately).

Parameters `name` : string

The display name of the CoordinateSym

frame : ReferenceFrame

The reference frame this base scalar belongs to

index : 0, 1 or 2

The index of the dimension denoted by this coordinate variable

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, CoordinateSym
>>> A = ReferenceFrame('A')
>>> A[1]
A_y
>>> type(A[0])
<class 'sympy.physics.vector.frame.CoordinateSym'>
```

(continues on next page)

(continued from previous page)

```
>>> a_y = CoordinateSym('a_y', A, 1)
>>> a_y == A[1]
True
```

ReferenceFrame

class sympy.physics.vector.frame.**ReferenceFrame**(name, indices=None, latexs=None, variables=None)

A reference frame in classical mechanics.

ReferenceFrame is a class used to represent a reference frame in classical mechanics. It has a standard basis of three unit vectors in the frame's x, y, and z directions.

It also can have a rotation relative to a parent frame; this rotation is defined by a direction cosine matrix relating this frame's basis vectors to the parent frame's basis vectors. It can also have an angular velocity vector, defined in another frame.

ang_acc_in(otherframe)

Returns the angular acceleration Vector of the ReferenceFrame.

Effectively returns the Vector: $\hat{N} \alpha \hat{B}$ which represent the angular acceleration of B in N, where B is self, and N is otherframe.

Parameters otherframe : ReferenceFrame

The ReferenceFrame which the angular acceleration is returned in.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

ang_vel_in(otherframe)

Returns the angular velocity Vector of the ReferenceFrame.

Effectively returns the Vector: $\hat{N} \omega \hat{B}$ which represent the angular velocity of B in N, where B is self, and N is otherframe.

Parameters otherframe : ReferenceFrame

The ReferenceFrame which the angular velocity is returned in.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
```

(continues on next page)

(continued from previous page)

```
>>> A.ang_vel_in(N)
10*N.x
```

dcm(otherframe)

The direction cosine matrix between frames.

This gives the DCM between this frame and the otherframe. The format is $N.xyz = N.dcm(B) * B.xyz$. A SymPy Matrix is returned.

Parameters otherframe : ReferenceFrame

The otherframe which the DCM is generated to.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> N.dcm(A)
Matrix([
[1, 0, 0],
[0, cos(q1), -sin(q1)],
[0, sin(q1), cos(q1)]])
```

orient(parent, rot_type, amounts, rot_order=)

Defines the orientation of this frame relative to a parent frame.

Parameters parent : ReferenceFrame

The frame that this ReferenceFrame will have its orientation matrix defined in relation to.

rot_type : str

The type of orientation matrix that is being created. Supported types are 'Body', 'Space', 'Quaternion', 'Axis', and 'DCM'. See examples for correct usage.

amounts : list OR value

The quantities that the orientation matrix will be defined by. In case of `rot_type='DCM'`, value must be a `sympy.matrices.MatrixBase` object (or subclasses of it).

rot_order : str

If applicable, the order of a series of rotations.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> from sympy import symbols, eye, ImmutableMatrix
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
```

Now we have a choice of how to implement the orientation. First is Body. Body orientation takes this reference frame through three successive simple rotations. Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> B.orient(N, 'Body', [q1, q2, q3], '123')
>>> B.orient(N, 'Body', [q1, q2, 0], 'ZXZ')
>>> B.orient(N, 'Body', [0, 0, 0], 'YXZ')
```

Next is Space. Space is like Body, but the rotations are applied in the opposite order.

```
>>> B.orient(N, 'Space', [q1, q2, q3], '312')
```

Next is Quaternion. This orients the new ReferenceFrame with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta. This orientation is described by four parameters: $q0 = \cos(\theta/2)$ $q1 = \lambda_x \sin(\theta/2)$ $q2 = \lambda_y \sin(\theta/2)$ $q3 = \lambda_z \sin(\theta/2)$ Quaternion does not take in a rotation order.

```
>>> B.orient(N, 'Quaternion', [q0, q1, q2, q3])
```

Next is Axis. This is a rotation about an arbitrary, non-time-varying axis by some angle. The axis is supplied as a Vector. This is how simple rotations are defined.

```
>>> B.orient(N, 'Axis', [q1, N.x + 2 * N.y])
```

Last is DCM (Direction Cosine Matrix). This is a rotation matrix given manually.

```
>>> B.orient(N, 'DCM', eye(3))
>>> B.orient(N, 'DCM', ImmutableMatrix([[0, 1, 0], [0, 0, -1], [-1, 0, 0]]))
```

orientnew(newname, rot_type, amounts, rot_order="", variables=None, indices=None, latexs=None)

Creates a new ReferenceFrame oriented with respect to this Frame.

See ReferenceFrame.orient() for acceptable rotation types, amounts, and orders. Parent is going to be self.

Parameters newname : str

The name for the new ReferenceFrame

rot_type : str

The type of orientation matrix that is being created.

amounts : list OR value

The quantities that the orientation matrix will be defined by.

rot_order : str

If applicable, the order of a series of rotations.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')
```


Now we have a choice of how to implement the orientation. First is Body. Body orientation takes this reference frame through three successive simple rotations. Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about an axis twice in a row.

```
>>> A = N.orientnew('A', 'Body', [q1, q2, q3], '123')
>>> A = N.orientnew('A', 'Body', [q1, q2, 0], 'ZXZ')
>>> A = N.orientnew('A', 'Body', [0, 0, 0], 'XYX')
```

Next is Space. Space is like Body, but the rotations are applied in the opposite order.

```
>>> A = N.orientnew('A', 'Space', [q1, q2, q3], '312')
```

Next is Quaternion. This orients the new ReferenceFrame with Quaternions, defined as a finite rotation about λ , a unit vector, by some amount θ . This orientation is described by four parameters: $q_0 = \cos(\theta/2)$ $q_1 = \lambda_x \sin(\theta/2)$ $q_2 = \lambda_y \sin(\theta/2)$ $q_3 = \lambda_z \sin(\theta/2)$ Quaternion does not take in a rotation order.

```
>>> A = N.orientnew('A', 'Quaternion', [q0, q1, q2, q3])
```

Last is Axis. This is a rotation about an arbitrary, non-time-varying axis by some angle. The axis is supplied as a Vector. This is how simple rotations are defined.

```
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
```

partial_velocity(frame, *gen_speeds)

Returns the partial angular velocities of this frame in the given frame with respect to one or more provided generalized speeds.

Parameters **frame** : ReferenceFrame

The frame with which the angular velocity is defined in.

gen_speeds : functions of time

The generalized speeds.

Returns **partial_velocities** : tuple of Vector

The partial angular velocity vectors corresponding to the provided generalized speeds.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> A.set_ang_vel(N, u1 * A.x + u2 * N.y)
>>> A.partial_velocity(N, u1)
A.x
>>> A.partial_velocity(N, u1, u2)
(A.x, N.y)
```

set_ang_acc(otherframe, value)

Define the angular acceleration Vector in a ReferenceFrame.

Defines the angular acceleration of this ReferenceFrame, in another. Angular acceleration can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

Parameters **otherframe** : ReferenceFrame

A ReferenceFrame to define the angular acceleration in

value : Vector

The Vector representing angular acceleration

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

set_ang_vel(otherframe, value)

Define the angular velocity vector in a ReferenceFrame.

Defines the angular velocity of this ReferenceFrame, in another. Angular velocity can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

Parameters **otherframe** : ReferenceFrame

A ReferenceFrame to define the angular velocity in

value : Vector

The Vector representing angular velocity

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
>>> A.ang_vel_in(N)
10*N.x
```

variable_map(otherframe)

Returns a dictionary which expresses the coordinate variables of this frame in terms of the variables of otherframe.

If Vector.simp is True, returns a simplified version of the mapped values. Else, returns them without simplification.

Simplification of the expressions may take time.

Parameters **otherframe** : ReferenceFrame

The other frame to map the variables to

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> A = ReferenceFrame('A')
>>> q = dynamicsymbols('q')
>>> B = A.orientnew('B', 'Axis', [q, A.z])
>>> A.variable_map(B)
{A_x: B_x*cos(q(t)) - B_y*sin(q(t)), A_y: B_x*sin(q(t)) + B_y*cos(q(t)), A_z:
↪ B_z}
```

x

The basis Vector for the ReferenceFrame, in the x direction.

y

The basis Vector for the ReferenceFrame, in the y direction.

z

The basis Vector for the ReferenceFrame, in the z direction.

Vector

class sympy.physics.vector.vector.**Vector**(inlist)

The class used to define vectors.

It along with ReferenceFrame are the building blocks of describing a classical mechanics system in PyDy and sympy.physics.vector.

Attributes

simp	(Boolean) Let certain methods use trigsimp on their outputs

applyfunc(f)

Apply a function to each component of a vector.

cross(other)

The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

Parameters **other** : Vector

The Vector which we are crossing with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> N.x ^ N.y
N.z
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.x ^ N.y
```

(continues on next page)

(continued from previous page)

```
N.z
>>> N.y ^ A.x
- sin(q1)*A.y - cos(q1)*A.z
```

diff(var, frame, var_in_dcm=True)

Returns the partial derivative of the vector with respect to a variable in the provided reference frame.

Parameters **var** : Symbol

What the partial derivative is taken with respect to.

frame : ReferenceFrame

The reference frame that the partial derivative is taken in.

var_in_dcm : boolean

If true, the differentiation algorithm assumes that the variable may be present in any of the direction cosine matrices that relate the frame to the frames of any component of the vector. But if it is known that the variable is not present in the direction cosine matrices, false can be set to skip full reexpression in the desired frame.

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.vector import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.vector import Vector
>>> Vector.simp = True
>>> t = Symbol('t')
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.diff(t, N)
- q1'*A.z
>>> B = ReferenceFrame('B')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> v = u1 * A.x + u2 * B.y
>>> v.diff(u2, N, var_in_dcm=False)
B.y
```

doit(**hints)

Calls .doit() on each term in the Vector

dot(other)

Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

Parameters **other** : Vector

The Vector which we are dotting with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

dt(otherframe)

Returns a Vector which is the time derivative of the self Vector, taken in frame otherframe.

Calls the global time_derivative method

Parameters otherframe : ReferenceFrame

The frame to calculate the time derivative in

express(otherframe, variables=False)

Returns a Vector equivalent to this one, expressed in otherframe. Uses the global express method.

Parameters otherframe : ReferenceFrame

The frame for this Vector to be described in

variables : boolean

If True, the coordinate symbols(if present) in this Vector are re-expressed in terms otherframe

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Vector, dynamicsymbols
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.express(N)
cos(q1)*N.x - sin(q1)*N.z
```

free_symbols(reference_frame)

Returns the free symbols in the measure numbers of the vector expressed in the given reference frame.

Parameter

reference_frame [ReferenceFrame] The frame with respect to which the free symbols of the given vector is to be determined.

magnitude()

Returns the magnitude (Euclidean norm) of self.

normalize()

Returns a Vector of magnitude 1, codirectional with self.

outer(other)

Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

Parameters **other** : Vector

The Vector to take the outer product with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

separate()

The constituents of this vector in different reference frames, as per its definition.

Returns a dict mapping each ReferenceFrame to the corresponding constituent Vector.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R1 = ReferenceFrame('R1')
>>> R2 = ReferenceFrame('R2')
>>> v = R1.x + R2.x
>>> v.separate() == {R1: R1.x, R2: R2.x}
True
```

simplify()

Returns a simplified Vector.

subs(*args, **kwargs)

Substitution on the Vector.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = N.x * s
>>> a.subs({s: 2})
2*N.x
```

to_matrix(reference_frame)

Returns the matrix form of the vector with respect to the given frame.

Parameters **reference_frame** : ReferenceFrame

The reference frame that the rows of the matrix correspond to.

Returns matrix : ImmutableMatrix, shape(3,1)

The matrix that gives the 1D vector.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.mechanics.functions import inertia
>>> a, b, c = symbols('a, b, c')
>>> N = ReferenceFrame('N')
>>> vector = a * N.x + b * N.y + c * N.z
>>> vector.to_matrix(N)
Matrix([
[a],
[b],
[c]])
>>> beta = symbols('beta')
>>> A = N.orientnew('A', 'Axis', (beta, N.x))
>>> vector.to_matrix(A)
Matrix([
[          a],
[ b*cos(beta) + c*sin(beta)],
[ -b*sin(beta) + c*cos(beta)]])
```

Dyadic

class sympy.physics.vector.dyadic.Dyadic(inlist)

A Dyadic object.

See: http://en.wikipedia.org/wiki/Dyadic_tensor Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

A more powerful way to represent a rigid body's inertia. While it is more complex, by choosing Dyadic components to be in body fixed basis vectors, the resulting matrix is equivalent to the inertia tensor.

applyfunc(f)

Apply a function to each component of a Dyadic.

cross(other)

For a cross product in the form: Dyadic x Vector.

Parameters other : Vector

The Vector that we are crossing this Dyadic with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, cross
>>> N = ReferenceFrame('N')
>>> d = outer(N.x, N.x)
>>> cross(d, N.y)
(N.x|N.z)
```

doit(**hints)

Calls `.doit()` on each term in the Dyadic

dot(other)

The inner product operator for a Dyadic and a Dyadic or Vector.

Parameters **other** : Dyadic or Vector

The other Dyadic or Vector to take the inner product with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> D1 = outer(N.x, N.y)
>>> D2 = outer(N.y, N.y)
>>> D1.dot(D2)
(N.x|N.y)
>>> D1.dot(N.y)
N.x
```

dt(frame)

Take the time derivative of this Dyadic in a frame.

This function calls the global `time_derivative` method

Parameters **frame** : ReferenceFrame

The frame to take the time derivative in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.dt(B)
- q'*(N.y|N.x) - q'*(N.x|N.y)
```

express(frame1, frame2=None)

Expresses this Dyadic in alternate frame(s)

The first frame is the list side expression, the second frame is the right side; if Dyadic is in form `A.x|B.y`, you can express it in two different frames. If no second frame is given, the Dyadic is expressed in only one frame.

Calls the global `express` function

Parameters **frame1** : ReferenceFrame

The frame to express the left side of the Dyadic in

frame2 : ReferenceFrame

If provided, the frame to express the right side of the Dyadic in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.express(B, N)
cos(q)*(B.x|N.x) - sin(q)*(B.y|N.x)
```

simplify()

Returns a simplified Dyadic.

subs(*args, **kwargs)

Substitution on the Dyadic.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = s * (N.x|N.x)
>>> a.subs({s: 2})
2*(N.x|N.x)
```

to_matrix(reference_frame, second_reference_frame=None)

Returns the matrix form of the dyadic with respect to one or two reference frames.

Parameters **reference_frame** : ReferenceFrame

The reference frame that the rows and columns of the matrix correspond to. If a second reference frame is provided, this only corresponds to the rows of the matrix.

second_reference_frame : ReferenceFrame, optional, default=None

The reference frame that the columns of the matrix correspond to.

Returns **matrix** : ImmutableMatrix, shape(3,3)

The matrix that gives the 2D tensor form.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> Vector.simp = True
>>> from sympy.physics.mechanics import inertia
>>> Ixx, Iyy, Izz, Ixy, Iyz, Ixz = symbols('Ixx, Iyy, Izz, Ixy, Iyz, Ixz')
>>> N = ReferenceFrame('N')
>>> inertia_dyadic = inertia(N, Ixx, Iyy, Izz, Ixy, Iyz, Ixz)
>>> inertia_dyadic.to_matrix(N)
Matrix([
  [Ixx, Ixy, Ixz],
  [Ixy, Iyy, Iyz],
  [Ixz, Iyz, Izz]])
```

(continues on next page)

(continued from previous page)

```
[Ixz, Iyz, Izz]])
>>> beta = symbols('beta')
>>> A = N.orientnew('A', 'Axis', (beta, N.x))
>>> inertia_dyadic.to_matrix(A)
Matrix([
[
      Ixx,
      ↪ Ixy*cos(beta) + Ixz*sin(beta),
      ↪ Ixy*sin(beta) + Ixz*cos(beta)],
[ Ixy*cos(beta) + Ixz*sin(beta), Iyy*cos(2*beta)/2 + Iyy/2 + Iyz*sin(2*beta) -
      ↪ Izz*cos(2*beta)/2 + Izz/2,
      ↪ Iyz*cos(2*beta) + Izz*sin(2*beta)/2],
[ -Ixy*sin(beta) + Ixz*cos(beta),
      ↪ Iyz*cos(2*beta) + Izz*sin(2*beta)/2, -Iyy*cos(2*beta)/2 + Iyy/2
      ↪ Iyz*sin(2*beta) + Izz*cos(2*beta)/2 + Izz/2]])
```

Kinematics (Docstrings)

Point

class `sympy.physics.vector.point.Point(name)`

This object represents a point in a dynamic system.

It stores the: position, velocity, and acceleration of a point. The position is a vector defined as the vector distance from a parent point to this point.

alpt_theory(otherpoint, outframe, interframe)

Sets the acceleration of this point with the 1-point theory.

The 1-point theory for point acceleration looks like this:

$${}^N a^P = {}^B a^P + {}^N a^O + {}^N \alpha^B \times r^{OP} + {}^N \omega^B \times ({}^N \omega^B \times r^{OP}) + 2 {}^N \omega^B \times {}^B v^P$$

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

Parameters **otherpoint** : Point

The first point of the 1-point theory (O)

outframe : ReferenceFrame

The frame we want this point's acceleration defined in (N)

fixedframe : ReferenceFrame

The intermediate frame in this calculation (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import Vector, dynamicsymbols
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
```

(continues on next page)

(continued from previous page)

```

>>> B.set_ang_vel(N, 5 * B.y)
>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.a1pt_theory(O, N, B)
(-25*q + q'')*B.x + q2''*B.y - 10*q'*B.z

```

a2pt_theory(otherpoint, outframe, fixedframe)

Sets the acceleration of this point with the 2-point theory.

The 2-point theory for point acceleration looks like this:

$${}^N a^P = {}^N a^O + {}^N \alpha^B \times r^{OP} + {}^N \omega^B \times ({}^N \omega^B \times r^{OP})$$

where O and P are both points fixed in frame B, which is rotating in frame N.

Parameters otherpoint : Point

The first point of the 2-point theory (O)

outframe : ReferenceFrame

The frame we want this point's acceleration defined in (N)

fixedframe : ReferenceFrame

The frame in which both points are fixed (B)

Examples

```

>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.a2pt_theory(O, N, B)
- 10*q''*2*B.x + 10*q'*B.y

```

acc(frame)

The acceleration Vector of this Point in a ReferenceFrame.

Parameters frame : ReferenceFrame

The frame in which the returned acceleration vector will be defined in

Examples

```

>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x

```

locatenew(name, value)

Creates a new point with a position defined from this point.

Parameters **name** : str

The name for the new point

value : Vector

The position of the new point relative to this point

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> N = ReferenceFrame('N')
>>> P1 = Point('P1')
>>> P2 = P1.locatenew('P2', 10 * N.x)
```

partial_velocity(frame, *gen_speeds)

Returns the partial velocities of the linear velocity vector of this point in the given frame with respect to one or more provided generalized speeds.

Parameters **frame** : ReferenceFrame

The frame with which the velocity is defined in.

gen_speeds : functions of time

The generalized speeds.

Returns **partial_velocities** : tuple of Vector

The partial velocity vectors corresponding to the provided generalized speeds.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> p = Point('p')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> p.set_vel(N, u1 * N.x + u2 * A.y)
>>> p.partial_velocity(N, u1)
N.x
>>> p.partial_velocity(N, u1, u2)
(N.x, A.y)
```

pos_from(otherpoint)

Returns a Vector distance between this Point and the other Point.

Parameters **otherpoint** : Point

The otherpoint we are locating this one relative to

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

set_acc(frame, value)

Used to set the acceleration of this Point in a ReferenceFrame.

Parameters **frame** : ReferenceFrame

The frame in which this point's acceleration is defined

value : Vector

The vector value of this point's acceleration in the frame

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

set_pos(otherpoint, value)

Used to set the position of this point w.r.t. another point.

Parameters **otherpoint** : Point

The other point which this point's location is defined relative to

value : Vector

The vector which defines the location of this point

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

set_vel(frame, value)

Sets the velocity Vector of this Point in a ReferenceFrame.

Parameters **frame** : ReferenceFrame

The frame in which this point's velocity is defined

value : Vector

The vector value of this point's velocity in the frame

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

v1pt_theory(otherpoint, outframe, interframe)

Sets the velocity of this point with the 1-point theory.

The 1-point theory for point velocity looks like this:

$${}^N \mathbf{v}^P = {}^B \mathbf{v}^P + {}^N \mathbf{v}^O + {}^N \boldsymbol{\omega}^B \times \mathbf{r}^{OP}$$

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

Parameters otherpoint : Point

The first point of the 2-point theory (O)

outframe : ReferenceFrame

The frame we want this point's velocity defined in (N)

interframe : ReferenceFrame

The intermediate frame in this calculation (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import Vector, dynamicsymbols
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.set_ang_vel(N, 5 * B.y)
>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.v1pt_theory(O, N, B)
q'*B.x + q2'*B.y - 5*q*B.z
```

v2pt_theory(otherpoint, outframe, fixedframe)

Sets the velocity of this point with the 2-point theory.

The 2-point theory for point velocity looks like this:

$${}^N \mathbf{v}^P = {}^N \mathbf{v}^O + {}^N \boldsymbol{\omega}^B \times \mathbf{r}^{OP}$$

where O and P are both points fixed in frame B, which is rotating in frame N.

Parameters otherpoint : Point

The first point of the 2-point theory (O)

outframe : ReferenceFrame

The frame we want this point's velocity defined in (N)

fixedframe : ReferenceFrame

The frame in which both points are fixed (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.v2pt_theory(O, N, B)
5*N.x + 10*q'*B.y
```

vel(frame)

The velocity Vector of this Point in the ReferenceFrame.

Parameters frame : ReferenceFrame

The frame in which the returned velocity vector will be defined in

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

kinematic_equations

`sympy.physics.vector.functions.kinematic_equations(speeds, coords, rot_type, rot_order="")`

Gives equations relating the qdot's to u's for a rotation type.

Supply rotation type and order as in orient. Speeds are assumed to be body-fixed; if we are defining the orientation of B in A using by rot_type, the angular velocity of B in A is assumed to be in the form: $\text{speed}[0]*B.x + \text{speed}[1]*B.y + \text{speed}[2]*B.z$

Parameters speeds : list of length 3

The body fixed angular velocity measure numbers.

coords : list of length 3 or 4

The coordinates used to define the orientation of the two frames.

rot_type : str

The type of rotation used to create the equations. Body, Space, or Quaternion only

rot_order : str

If applicable, the order of a series of rotations.

Examples

```
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import kinematic_equations, vprint
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> vprint(kinematic_equations([u1,u2,u3], [q1,q2,q3], 'body', '313'),
...        order=None)
[-(u1*sin(q3) + u2*cos(q3))/sin(q2) + q1', -u1*cos(q3) + u2*sin(q3) + q2',
-(u1*sin(q3) + u2*cos(q3))*cos(q2)/sin(q2) - u3 + q3']
```

`sympy.physics.vector.functions.partial_velocity(vel_vecs, gen_speeds, frame)`

Returns a list of partial velocities with respect to the provided generalized speeds in the given reference frame for each of the supplied velocity vectors.

The output is a list of lists. The outer list has a number of elements equal to the number of supplied velocity vectors. The inner lists are, for each velocity vector, the partial derivatives of that velocity vector with respect to the generalized speeds supplied.

Parameters **vel_vecs** : iterable

An iterable of velocity vectors (angular or linear).

gen_speeds : iterable

An iterable of generalized speeds.

frame : ReferenceFrame

The reference frame that the partial derivatives are going to be taken in.

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import partial_velocity
>>> u = dynamicsymbols('u')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> vel_vecs = [P.vel(N)]
>>> gen_speeds = [u]
>>> partial_velocity(vel_vecs, gen_speeds, N)
[[N.x]]
```

`sympy.physics.vector.functions.get_motion_params(frame, **kwargs)`

Returns the three motion parameters - (acceleration, velocity, and position) as vectorial functions of time in the given frame.

If a higher order differential function is provided, the lower order functions are used as boundary conditions. For example, given the acceleration, the velocity and position parameters are taken as boundary conditions.

The values of time at which the boundary conditions are specified are taken from `timevalue1` (for position boundary condition) and `timevalue2` (for velocity boundary condition).

If any of the boundary conditions are not provided, they are taken to be zero by default (zero vectors, in case of vectorial inputs). If the boundary conditions are also functions of time, they are converted to constants by substituting the time values in the dynamic-symbols._t time Symbol.

This function can also be used for calculating rotational motion parameters. Have a look at the Parameters and Examples for more clarity.

Parameters **frame** : ReferenceFrame

The frame to express the motion parameters in

acceleration : Vector

Acceleration of the object/frame as a function of time

velocity : Vector

Velocity as function of time or as boundary condition of velocity at `time = timevalue1`

position : Vector

Velocity as function of time or as boundary condition of velocity at `time = timevalue1`

timevalue1 : sympyifiable

Value of time for position boundary condition

timevalue2 : sympyifiable

Value of time for velocity boundary condition

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, get_motion_params, \
↳ dynamicsymbols
>>> from sympy import symbols
>>> R = ReferenceFrame('R')
>>> v1, v2, v3 = dynamicsymbols('v1 v2 v3')
>>> v = v1*R.x + v2*R.y + v3*R.z
>>> get_motion_params(R, position = v)
(v1''*R.x + v2''*R.y + v3''*R.z, v1'*R.x + v2'*R.y + v3'*R.z, v1*R.x + v2*R.y + \
↳ v3*R.z)
>>> a, b, c = symbols('a b c')
>>> v = a*R.x + b*R.y + c*R.z
>>> get_motion_params(R, velocity = v)
(0, a*R.x + b*R.y + c*R.z, a*t*R.x + b*t*R.y + c*t*R.z)
>>> parameters = get_motion_params(R, acceleration = v)
>>> parameters[1]
a*t*R.x + b*t*R.y + c*t*R.z
>>> parameters[2]
a*t**2/2*R.x + b*t**2/2*R.y + c*t**2/2*R.z
```

Printing (Docstrings)

init_printing

```
sympy.physics.vector.printing.init_printing(pretty_print=True,          or-
                                             der=None,      use_unicode=None,
                                             use_latex=None, wrap_line=None,
                                             num_columns=None,
                                             no_global=False, ip=None,    eu-
                                             ler=False,      forecolor='Black',
                                             backcolor='Transparent',
                                             fontsize='10pt',              la-
                                             tex_mode='equation*',
                                             print_builtin=True,
                                             str_printer=None,
                                             pretty_printer=None,          la-
                                             tex_printer=None, **settings)
```

Initializes pretty-printer depending on the environment.

Parameters pretty_print: boolean

If True, use pretty_print to stringify or the provided pretty printer; if False, use sstrrepr to stringify or the provided string printer.

order: string or None

There are a few different settings for this parameter: lex (default), which is lexographic order; grlex, which is graded lexographic order; grevlex, which is reversed graded lexographic order; old, which is used for compatibility reasons and for long expressions; None, which sets it to lex.

use_unicode: boolean or None

If True, use unicode characters; if False, do not use unicode characters.

use_latex: string, boolean, or None

If True, use default latex rendering in GUI interfaces (png and mathjax); if False, do not use latex rendering; if 'png', enable latex rendering with an external latex compiler, falling back to matplotlib if external compilation fails; if 'matplotlib', enable latex rendering with matplotlib; if 'mathjax', enable latex text generation, for example MathJax rendering in IPython notebook or text rendering in LaTeX documents

wrap_line: boolean

If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if *pretty_print* is True.

num_columns: int or None

If int, number of columns before wrapping is set to num_columns; if None, number of columns before wrapping is set to terminal width. This is only relevant if *pretty_print* is True.

no_global: boolean

If True, the settings become system wide; if False, use just for this console/session.

ip: An interactive console

This can either be an instance of IPython, or a class that derives from `code.InteractiveConsole`.

euler: boolean, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (<http://www.ctan.org/pkg/euler>).

forecolor: string, optional, default='Black'

DVI setting for foreground color.

backcolor: string, optional, default='Transparent'

DVI setting for background color.

fontsize: string, optional, default='10pt'

A font size to pass to the LaTeX documentclass function in the preamble.

latex_mode: string, optional, default='equation*'

The mode used in the LaTeX printer. Can be one of: {'inline'|'plain'|'equation'|'equation*'}.

print_builtin: boolean, optional, default=True

If true then floats and integers will be printed. If false the printer will only print SymPy types.

str_printer: function, optional, default=None

A custom string printer function. This should mimic `sympy.printing.sstrrepr()`.

pretty_printer: function, optional, default=None

A custom pretty printer. This should mimic `sympy.printing.pretty()`.

latex_printer: function, optional, default=None

A custom LaTeX printer. This should mimic `sympy.printing.latex()`.

Examples

```
>>> from sympy.interactive import init_printing
>>> from sympy import Symbol, sqrt
>>> from sympy.abc import x, y
>>> sqrt(5)
sqrt(5)
>>> init_printing(pretty_print=True)
>>> sqrt(5)

$$\sqrt{5}$$

>>> theta = Symbol('theta')
>>> init_printing(use_unicode=True)
>>> theta
 $\theta$ 
>>> init_printing(use_unicode=False)
>>> theta
theta
```

(continues on next page)

(continued from previous page)

```

>>> init_printing(order='lex')
>>> str(y + x + y**2 + x**2)
x**2 + x + y**2 + y
>>> init_printing(order='grlex')
>>> str(y + x + y**2 + x**2)
x**2 + x + y**2 + y
>>> init_printing(order='grevlex')
>>> str(y * x**2 + x * y**2)
x**2*y + x*y**2
>>> init_printing(order='old')
>>> str(x**2 + y**2 + x + y)
x**2 + x + y**2 + y
>>> init_printing(num_columns=10)
>>> x**2 + x + y**2 + y
x + y +
x**2 + y**2

```

vprint

`sympy.physics.vector.printing.vprint(expr, **settings)`

Function for printing of expressions generated in the `sympy.physics` vector package.

Extends SymPy's `StrPrinter`, takes the same setting accepted by SymPy's `sstr()`, and is equivalent to `print(sstr(foo))`.

Parameters `expr` : valid SymPy object

SymPy expression to print.

settings : args

Same as the settings accepted by SymPy's `sstr()`.

Examples

```

>>> from sympy.physics.vector import vprint, dynamicsymbols
>>> u1 = dynamicsymbols('u1')
>>> print(u1)
u1(t)
>>> vprint(u1)
u1

```

vpprint

`sympy.physics.vector.printing.vpprint(expr, **settings)`

Function for pretty printing of expressions generated in the `sympy.physics` vector package.

Mainly used for expressions not inside a vector; the output of running scripts and generating equations of motion. Takes the same options as SymPy's `pretty_print()`; see that function for more information.

Parameters `expr` : valid SymPy object

SymPy expression to pretty print

settings : args

Same as those accepted by SymPy's `pretty_print`.

vlatex

`sympy.physics.vector.printing.vlatex(expr, **settings)`

Function for printing latex representation of `sympy.physics.vector` objects.

For latex representation of Vectors, Dyadics, and dynamicsymbols. Takes the same options as SymPy's `latex()`; see that function for more information;

Parameters **expr** : valid SymPy object

SymPy expression to represent in LaTeX form

settings : args

Same as `latex()`

Examples

```
>>> from sympy.physics.vector import vlatex, ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> q1dd, q2dd = dynamicsymbols('q1 q2', 2)
>>> vlatex(N.x + N.y)
'\mathbf{\hat{n}_x} + \mathbf{\hat{n}_y}'
>>> vlatex(q1 + q2)
'q_{1} + q_{2}'
>>> vlatex(q1d)
'\dot{q}_{1}'
>>> vlatex(q1 * q2d)
'q_{1} \dot{q}_{2}'
>>> vlatex(q1dd * q1 / q1d)
'\frac{q_{1} \ddot{q}_{1}}{\dot{q}_{1}}'
```

Essential Functions (Docstrings)

dynamicsymbols

`sympy.physics.vector.dynamicsymbols(names, level=0)`

Uses symbols and Function for functions of time.

Creates a SymPy UndefinedFunction, which is then initialized as a function of a variable, the default being `Symbol('t')`.

Parameters **names** : str

Names of the dynamic symbols you want to create; works the same way as inputs to symbols

level : int

Level of differentiation of the returned function; d/dt once of t, twice of t, etc.

Examples

```
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy import diff, Symbol
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)
```

dot

`sympy.physics.vector.functions.dot(vec1, vec2)`

Dot product convenience wrapper for `Vector.dot()`: Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

Parameters **other** : Vector

The Vector which we are dotting with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

cross

`sympy.physics.vector.functions.cross(vec1, vec2)`

Cross product convenience wrapper for `Vector.cross()`: The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

Parameters **other** : Vector

The Vector which we are crossing with

Examples

```

>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> N.x ^ N.y
N.z
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.x ^ N.y
N.z
>>> N.y ^ A.x
- sin(q1)*A.y - cos(q1)*A.z

```

outer

`sympy.physics.vector.functions.outer(vec1, vec2)`

Outer product convenience wrapper for `Vector.outer()`: Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

Parameters other : Vector

The Vector to take the outer product with

Examples

```

>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)

```

express

`sympy.physics.vector.functions.express(expr, frame, frame2=None, variables=False)`

Global function for 'express' functionality.

Re-expresses a Vector, scalar(sympyifiable) or Dyadic in given frame.

Refer to the local methods of Vector and Dyadic for details. If 'variables' is True, then the coordinate variables (CoordinateSym instances) of other frames present in the vector/scalar field or dyadic expression are also substituted in terms of the base scalars of this frame.

Parameters expr : Vector/Dyadic/scalar(sympyifiable)

The expression to re-express in ReferenceFrame 'frame'

frame: ReferenceFrame

The reference frame to express expr in

frame2 : ReferenceFrame

The other frame required for re-expression(only for Dyadic expr)

variables : boolean

Specifies whether to substitute the coordinate variables present in `expr`, in terms of those of frame

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> from sympy.physics.vector import express
>>> express(d, B, N)
cos(q)*(B.x|N.x) - sin(q)*(B.y|N.x)
>>> express(B.x, N)
cos(q)*N.x + sin(q)*N.y
>>> express(N[0], B, variables=True)
B_x*cos(q(t)) - B_y*sin(q(t))
```

time_derivative

`sympy.physics.vector.functions.time_derivative(expr, frame, order=1)`

Calculate the time derivative of a vector/scalar field function or dyadic expression in given frame.

Parameters `expr` : Vector/Dyadic/sympifyable

The expression whose time derivative is to be calculated

frame : ReferenceFrame

The reference frame to calculate the time derivative in

order : integer

The order of the derivative to be calculated

References

http://en.wikipedia.org/wiki/Rotating_reference_frame#Time_derivatives_in_the_two_frames

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> from sympy import Symbol
>>> q1 = Symbol('q1')
>>> u1 = dynamicsymbols('u1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> v = u1 * N.x
>>> A.set_ang_vel(N, 10*A.x)
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(v, N)
```

(continues on next page)

(continued from previous page)

```

u1'*N.x
>>> time_derivative(u1*A[0], N)
N_x*Derivative(u1(t), t)
>>> B = N.orientnew('B', 'Axis', [u1, N.z])
>>> from sympy.physics.vector import outer
>>> d = outer(N.x, N.x)
>>> time_derivative(d, B)
- u1'*(N.y|N.x) - u1'*(N.x|N.y)

```

Docstrings for basic field functions

Field operation functions

These functions implement some basic operations pertaining to fields in general.

curl

`sympy.physics.vector.fieldfunctions.curl(vect, frame)`

Returns the curl of a vector field computed wrt the coordinate symbols of the given frame.

Parameters `vect` : Vector

The vector operand

frame : ReferenceFrame

The reference frame to calculate the curl in

Examples

```

>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import curl
>>> R = ReferenceFrame('R')
>>> v1 = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> curl(v1, R)
0
>>> v2 = R[0]*R[1]*R[2]*R.x
>>> curl(v2, R)
R_x*R_y*R.y - R_x*R_z*R.z

```

divergence

`sympy.physics.vector.fieldfunctions.divergence(vect, frame)`

Returns the divergence of a vector field computed wrt the coordinate symbols of the given frame.

Parameters `vect` : Vector

The vector operand

frame : ReferenceFrame

The reference frame to calculate the divergence in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import divergence
>>> R = ReferenceFrame('R')
>>> v1 = R[0]*R[1]*R[2] * (R.x+R.y+R.z)
>>> divergence(v1, R)
R_x*R_y + R_x*R_z + R_y*R_z
>>> v2 = 2*R[1]*R[2]*R.y
>>> divergence(v2, R)
2*R_z
```

gradient

`sympy.physics.vector.fieldfunctions.gradient`(scalar, frame)

Returns the vector gradient of a scalar field computed wrt the coordinate symbols of the given frame.

Parameters **scalar** : sympifiable

The scalar field to take the gradient of

frame : ReferenceFrame

The frame to calculate the gradient in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import gradient
>>> R = ReferenceFrame('R')
>>> s1 = R[0]*R[1]*R[2]
>>> gradient(s1, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_x*R_y*R.z
>>> s2 = 5*R[0]**2*R[2]
>>> gradient(s2, R)
10*R_x*R_z*R.x + 5*R_x**2*R.z
```

scalar_potential

`sympy.physics.vector.fieldfunctions.scalar_potential`(field, frame)

Returns the scalar potential function of a field in a given frame (without the added integration constant).

Parameters **field** : Vector

The vector field whose scalar potential function is to be calculated

frame : ReferenceFrame

The frame to do the calculation in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import scalar_potential, gradient
>>> R = ReferenceFrame('R')
>>> scalar_potential(R.z, R) == R[2]
True
>>> scalar_field = 2*R[0]**2*R[1]*R[2]
>>> grad_field = gradient(scalar_field, R)
>>> scalar_potential(grad_field, R)
2*R_x**2*R_y*R_z
```

scalar_potential_difference

`sympy.physics.vector.fieldfunctions.scalar_potential_difference`(field,
frame,
point1,
point2,
origin)

Returns the scalar potential difference between two points in a certain frame, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at position 2) - (potential at position 1)

Parameters **field** : Vector/sympyifiable

The field to calculate wrt

frame : ReferenceFrame

The frame to do the calculations in

point1 : Point

The initial Point in given frame

position2 : Point

The second Point in the given frame

origin : Point

The Point to use as reference point for position vector calculation

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', R[0]*R.x + R[1]*R.y + R[2]*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
2*R_x**2*R_y
```

(continues on next page)

(continued from previous page)

```
>>> Q = 0.locatenew('0', 3*R.x + R.y + 2*R.z)
>>> scalar_potential_difference(vectfield, R, P, Q, 0)
-2*R_x**2*R_y + 18
```

Checking the type of vector field

is_conservative

`sympy.physics.vector.fieldfunctions.is_conservative(field)`

Checks if a field is conservative.

Parameters **field** : Vector

The field to check for conservative property

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_conservative
>>> R = ReferenceFrame('R')
>>> is_conservative(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_conservative(R[2] * R.y)
False
```

is_solenoidal

`sympy.physics.vector.fieldfunctions.is_solenoidal(field)`

Checks if a field is solenoidal.

Parameters **field** : Vector

The field to check for solenoidal property

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_solenoidal
>>> R = ReferenceFrame('R')
>>> is_solenoidal(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_solenoidal(R[1] * R.y)
False
```

Classical Mechanics

Abstract

In this documentation many components of the physics/mechanics module will be discussed. `mechanics` has been written to allow for creation of symbolic equations of motion for complicated multibody systems.

Vector

This module derives the vector-related abilities and related functionalities from `physics.vector`. Please have a look at the documentation of `physics.vector` and its necessary API to understand the vector capabilities of `mechanics`.

Mechanics

In physics, mechanics describes conditions of rest (statics) or motion (dynamics). There are a few common steps to all mechanics problems. First, an idealized representation of a system is described. Next, we use physical laws to generate equations that define the system's behavior. Then, we solve these equations, sometimes analytically but usually numerically. Finally, we extract information from these equations and solutions. The current scope of the module is multi-body dynamics: the motion of systems of multiple particles and/or rigid bodies. For example, this module could be used to understand the motion of a double pendulum, planets, robotic manipulators, bicycles, and any other system of rigid bodies that may fascinate us.

Often, the objective in multi-body dynamics is to obtain the trajectory of a system of rigid bodies through time. The challenge for this task is to first formulate the equations of motion of the system. Once they are formulated, they must be solved, that is, integrated forward in time. When digital computers came around, solving became the easy part of the problem. Now, we can tackle more complicated problems, which leaves the challenge of formulating the equations.

The term "equations of motion" is used to describe the application of Newton's second law to multi-body systems. The form of the equations of motion depends on the method used to generate them. This package implements two of these methods: Kane's method and Lagrange's method. This module facilitates the formulation of equations of motion, which can then be solved (integrated) using generic ordinary differential equation (ODE) solvers.

The approach to a particular class of dynamics problems, that of forward dynamics, has the following steps:

1. describing the system's geometry and configuration,
2. specifying the way the system can move, including constraints on its motion
3. describing the external forces and moments on the system,
4. combining the above information according to Newton's second law ($\mathbf{F} = m\mathbf{a}$), and
5. organizing the resulting equations so that they can be integrated to obtain the system's trajectory through time.

Together with the rest of SymPy, this module performs steps 4 and 5, provided that the user can perform 1 through 3 for the module. That is to say, the user must provide a complete representation of the free body diagrams that themselves represent the system, with which this code can provide equations of motion in a form amenable to numerical integration. Step 5 above amounts to arduous algebra for even fairly simple multi-body systems. Thus, it is desirable to use a symbolic math package, such as SymPy, to perform this step. It is for

this reason that this module is a part of SymPy. Step 4 amounts to this specific module, `sympy.physics.mechanics`.

Guide to Mechanics

Masses, Inertias, Particles and Rigid Bodies in Physics/Mechanics

This document will describe how to represent masses and inertias in mechanics and use of the `RigidBody` and `Particle` classes.

It is assumed that the reader is familiar with the basics of these topics, such as finding the center of mass for a system of particles, how to manipulate an inertia tensor, and the definition of a particle and rigid body. Any advanced dynamics text can provide a reference for these details.

Mass

The only requirement for a mass is that it needs to be a sympify-able expression. Keep in mind that masses can be time varying.

Particle

Particles are created with the class `Particle` in mechanics. A `Particle` object has an associated point and an associated mass which are the only two attributes of the object.:

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> m = Symbol('m')
>>> po = Point('po')
>>> # create a particle container
>>> pa = Particle('pa', po, m)
```

The associated point contains the position, velocity and acceleration of the particle. mechanics allows one to perform kinematic analysis of points separate from their association with masses.

Inertia

See the Inertia (Dyadics) section in 'Advanced Topics' part of physics/vector docs.

Rigid Body

Rigid bodies are created in a similar fashion as particles. The `RigidBody` class generates objects with four attributes: mass, center of mass, a reference frame, and an inertia tuple:

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> from sympy.physics.mechanics import outer
>>> m = Symbol('m')
```

(continues on next page)

(continued from previous page)

```

>>> A = ReferenceFrame('A')
>>> P = Point('P')
>>> I = outer(A.x, A.x)
>>> # create a rigid body
>>> B = RigidBody('B', P, A, m, (I, P))

```

The mass is specified exactly as is in a particle. Similar to the Particle's `.point`, the RigidBody's center of mass, `.masscenter` must be specified. The reference frame is stored in an analogous fashion and holds information about the body's orientation and angular velocity. Finally, the inertia for a rigid body needs to be specified about a point. In mechanics, you are allowed to specify any point for this. The most common is the center of mass, as shown in the above code. If a point is selected which is not the center of mass, ensure that the position between the point and the center of mass has been defined. The inertia is specified as a tuple of length two with the first entry being a Dyadic and the second entry being a Point of which the inertia dyadic is defined about.

Dyadic

In mechanics, dyadics are used to represent inertia ([Kane1985] (page 1873), [WikiDyadics] (page 1872), [WikiDyadicProducts] (page 1873)). A dyadic is a linear polynomial of component unit dyadics, similar to a vector being a linear polynomial of component unit vectors. A dyadic is the outer product between two vectors which returns a new quantity representing the juxtaposition of these two vectors. For example:

$$\begin{aligned}\hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{a}}_{\mathbf{x}} &= \hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{x}} \\ \hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{a}}_{\mathbf{y}} &= \hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{y}}\end{aligned}$$

Where $\hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{x}}$ and $\hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{a}}_{\mathbf{y}}$ are the outer products obtained by multiplying the left side as a column vector by the right side as a row vector. Note that the order is significant.

Some additional properties of a dyadic are:

$$\begin{aligned}(x\mathbf{v}) \otimes \mathbf{w} &= \mathbf{v} \otimes (x\mathbf{w}) = x(\mathbf{v} \otimes \mathbf{w}) \\ \mathbf{v} \otimes (\mathbf{w} + \mathbf{u}) &= \mathbf{v} \otimes \mathbf{w} + \mathbf{v} \otimes \mathbf{u} \\ (\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} &= \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u}\end{aligned}$$

A vector in a reference frame can be represented as $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ or $a\hat{\mathbf{i}} + b\hat{\mathbf{j}} + c\hat{\mathbf{k}}$. Similarly, a dyadic can be represented in tensor form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

or in dyadic form:

$$a_{11}\hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{x}} + a_{12}\hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{y}} + a_{13}\hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{z}} + a_{21}\hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{x}} + a_{22}\hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{y}} + a_{23}\hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{z}} + a_{31}\hat{\mathbf{a}}_{\mathbf{z}}\hat{\mathbf{a}}_{\mathbf{x}} + a_{32}\hat{\mathbf{a}}_{\mathbf{z}}\hat{\mathbf{a}}_{\mathbf{y}} + a_{33}\hat{\mathbf{a}}_{\mathbf{z}}\hat{\mathbf{a}}_{\mathbf{z}}$$

Just as with vectors, the later representation makes it possible to keep track of which frames the dyadic is defined with respect to. Also, the two components of each term in the dyadic need not be in the same frame. The following is valid:

$$\hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{b}}_{\mathbf{y}} = \hat{\mathbf{a}}_{\mathbf{x}} \hat{\mathbf{b}}_{\mathbf{y}}$$

Dyadics can also be crossed and dotted with vectors; again, order matters:

$$\begin{aligned}
 \hat{\mathbf{a}}_x \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_x \\
 \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_y \\
 \hat{\mathbf{a}}_x \hat{\mathbf{a}}_y \cdot \hat{\mathbf{a}}_x &= 0 \\
 \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_x \\
 \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_x \hat{\mathbf{a}}_y &= \hat{\mathbf{a}}_y \\
 \hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x &= 0 \\
 \hat{\mathbf{a}}_x \times \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x &= \hat{\mathbf{a}}_z \hat{\mathbf{a}}_x \\
 \hat{\mathbf{a}}_x \times \hat{\mathbf{a}}_x \hat{\mathbf{a}}_x &= 0 \\
 \hat{\mathbf{a}}_y \hat{\mathbf{a}}_x \times \hat{\mathbf{a}}_z &= -\hat{\mathbf{a}}_y \hat{\mathbf{a}}_y
 \end{aligned}$$

One can also take the time derivative of dyadics or express them in different frames, just like with vectors.

Linear Momentum

The linear momentum of a particle P is defined as:

$$L_P = m\mathbf{v}$$

where m is the mass of the particle P and $\mathbf{v}$ is the velocity of the particle in the inertial frame.[Likins1973]_.

Similarly the linear momentum of a rigid body is defined as:

$$L_B = m\mathbf{v}^*$$

where m is the mass of the rigid body, B, and $\mathbf{v}^*$ is the velocity of the mass center of B in the inertial frame.

Angular Momentum

The angular momentum of a particle P about an arbitrary point O in an inertial frame N is defined as:

$${}^N\mathbf{H}^{P/O} = \mathbf{r} \times m\mathbf{v}$$

where $\mathbf{r}$ is a position vector from point O to the particle of mass m and $\mathbf{v}$ is the velocity of the particle in the inertial frame.

Similarly the angular momentum of a rigid body B about a point O in an inertial frame N is defined as:

$${}^N\mathbf{H}^{B/O} = {}^N\mathbf{H}^{B/B^*} + {}^N\mathbf{H}^{B^*/O}$$

where the angular momentum of the body about it's mass center is:

$${}^N\mathbf{H}^{B/B^*} = \mathbf{I}^* \cdot \omega$$

and the angular momentum of the mass center about O is:

$${}^N\mathbf{H}^{B^*/O} = \mathbf{r}^* \times m\mathbf{v}^*$$

where $\mathbf{I}^*$ is the central inertia dyadic of rigid body B, ω is the inertial angular velocity of B, $\mathbf{r}^*$ is a position vector from point O to the mass center of B, m is the mass of B and $\mathbf{v}^*$ is the velocity of the mass center in the inertial frame.

Using momenta functions in Mechanics

The following example shows how to use the momenta functions in mechanics.

One begins by creating the requisite symbols to describe the system. Then the reference frame is created and the kinematics are done.

```
>> from sympy import symbols
>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
>> from sympy.physics.mechanics import RigidBody, Particle, Point, outer
>> from sympy.physics.mechanics import linear_momentum, angular_momentum
>> m, M, l1 = symbols('m M l1')
>> q1d = dynamicsymbols('q1d')
>> N = ReferenceFrame('N')
>> O = Point('O')
>> O.set_vel(N, 0 * N.x)
>> Ac = O.locatenew('Ac', l1 * N.x)
>> P = Ac.locatenew('P', l1 * N.x)
>> a = ReferenceFrame('a')
>> a.set_ang_vel(N, q1d * N.z)
>> Ac.v2pt_theory(O, N, a)
>> P.v2pt_theory(O, N, a)
```

Finally, the bodies that make up the system are created. In this case the system consists of a particle Pa and a RigidBody A.

```
>> Pa = Particle('Pa', P, m)
>> I = outer(N.z, N.z)
>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

Then one can either choose to evaluate the momenta of individual components of the system or of the entire system itself.

```
>> linear_momentum(N,A)
M*l1*q1d*N.y
>> angular_momentum(O, N, Pa)
4*l1**2*m*q1d*N.z
>> linear_momentum(N, A, Pa)
(M*l1*q1d + 2*l1*m*q1d)*N.y
>> angular_momentum(O, N, A, Pa)
(4*l1**2*m*q1d + q1d)*N.z
```

It should be noted that the user can determine either momenta in any frame in mechanics as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial linear and angular momenta. Please refer to the docstrings on each function to learn more about how each function works precisely.

Kinetic Energy

The kinetic energy of a particle P is defined as

$$T_P = \frac{1}{2}m\mathbf{v}^2$$

where m is the mass of the particle P and $\mathbf{v}$ is the velocity of the particle in the inertial frame.

Similarly the kinetic energy of a rigid body B is defined as

$$T_B = T_t + T_r$$

where the translational kinetic energy is given by:

$$T_t = \frac{1}{2} m \mathbf{v}^* \cdot \mathbf{v}^*$$

and the rotational kinetic energy is given by:

$$T_r = \frac{1}{2} \omega \cdot \mathbf{I}^* \cdot \omega$$

where m is the mass of the rigid body, $\mathbf{v}^*$ is the velocity of the mass center in the inertial frame, ω is the inertial angular velocity of the body and $\mathbf{I}^*$ is the central inertia dyadic.

Potential Energy

Potential energy is defined as the energy possessed by a body or system by virtue of its position or arrangement.

Since there are a variety of definitions for potential energy, this is not discussed further here. One can learn more about this in any elementary text book on dynamics.

Lagrangian

The Lagrangian of a body or a system of bodies is defined as:

$$\mathcal{L} = T - V$$

where T and V are the kinetic and potential energies respectively.

Using energy functions in Mechanics

The following example shows how to use the energy functions in mechanics.

As was discussed above in the momenta functions, one first creates the system by going through an identical procedure.

```
>> from sympy import symbols
>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame, outer
>> from sympy.physics.mechanics import RigidBody, Particle, mechanics_printing
>> from sympy.physics.mechanics import kinetic_energy, potential_energy, Point
>> mechanics_printing()
>> m, M, l1, g, h, H = symbols('m M l1 g h H')
>> omega = dynamicsymbols('omega')
>> N = ReferenceFrame('N')
>> O = Point('O')
>> O.set_vel(N, 0 * N.x)
>> Ac = O.locatenew('Ac', l1 * N.x)
>> P = Ac.locatenew('P', l1 * N.x)
>> a = ReferenceFrame('a')
>> a.set_ang_vel(N, omega * N.z)
>> Ac.v2pt_theory(O, N, a)
>> P.v2pt_theory(O, N, a)
>> Pa = Particle('Pa', P, m)
>> I = outer(N.z, N.z)
>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

The user can then determine the kinetic energy of any number of entities of the system:

```
>> kinetic_energy(N, Pa)
2*l1**2*m*qld**2
>> kinetic_energy(N, Pa, A)
M*l1**2*qld**2/2 + 2*l1**2*m*qld**2 + qld**2/2
```

It should be noted that the user can determine either kinetic energy relative to any frame in mechanics as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial kinetic energy.

For potential energies, the user must first specify the potential energy of every entity of the system using the `potential_energy` property. The potential energy of any number of entities comprising the system can then be determined:

```
>> Pa.potential_energy = m * g * h
>> A.potential_energy = M * g * H
>> potential_energy(A, Pa)
H*M*g + g*h*m
```

One can also determine the Lagrangian for this system:

```
>> Lagrangian(Pa, A)
-H*M*g + M*l1**2*qld**2/2 - g*h*m + 2*l1**2*m*qld**2 + qld**2/2
```

Please refer to the docstrings to learn more about each function.

Kane's Method in Physics/Mechanics

`mechanics` provides functionality for deriving equations of motion using Kane's method [Kane1985] (page 1873). This document will describe Kane's method as used in this module, but not how the equations are actually derived.

Structure of Equations

In mechanics we are assuming there are 5 basic sets of equations needed to describe a system. They are: holonomic constraints, non-holonomic constraints, kinematic differential equations, dynamic equations, and differentiated non-holonomic equations.

$$\begin{aligned}\mathbf{f}_h(q, t) &= 0 \\ \mathbf{k}_{nh}(q, t)u + \mathbf{f}_{nh}(q, t) &= 0 \\ \mathbf{k}_{kq}(q, t)\dot{q} + \mathbf{k}_{ku}(q, t)u + \mathbf{f}_k(q, t) &= 0 \\ \mathbf{k}_d(q, t)\dot{u} + \mathbf{f}_d(q, \dot{q}, u, t) &= 0 \\ \mathbf{k}_{dnh}(q, t)\dot{u} + \mathbf{f}_{dnh}(q, \dot{q}, u, t) &= 0\end{aligned}$$

In mechanics holonomic constraints are only used for the linearization process; it is assumed that they will be too complicated to solve for the dependent coordinate(s). If you are able to easily solve a holonomic constraint, you should consider redefining your problem in terms of a smaller set of coordinates. Alternatively, the time-differentiated holonomic constraints can be supplied.

Kane's method forms two expressions, F_r and F_r^* , whose sum is zero. In this module, these expressions are rearranged into the following form:

$$\mathbf{M}(q, t)\dot{u} = \mathbf{f}(q, \dot{q}, u, t)$$

For a non-holonomic system with o total speeds and m motion constraints, we will get $o - m$ equations. The mass-matrix/forcing equations are then augmented in the following fashion:

$$\mathbf{M}(q, t) = \begin{bmatrix} \mathbf{k}_d(q, t) \\ \mathbf{k}_{dnh}(q, t) \end{bmatrix}$$

$$(\text{forcing})(q, \dot{q}, u, t) = \begin{bmatrix} -\mathbf{f}_d(q, \dot{q}, u, t) \\ -\mathbf{f}_{dnh}(q, \dot{q}, u, t) \end{bmatrix}$$

Kane's Method in Physics/Mechanics

The formulation of the equations of motion in mechanics starts with creation of a `KanesMethod` object. Upon initialization of the `KanesMethod` object, an inertial reference frame needs to be supplied. along with some basic system information, such as coordinates and speeds

```
>>> from sympy.physics.mechanics import *
>>> N = ReferenceFrame('N')
>>> q1, q2, u1, u2 = dynamicsymbols('q1 q2 u1 u2')
>>> q1d, q2d, u1d, u2d = dynamicsymbols('q1 q2 u1 u2', 1)
>>> KM = KanesMethod(N, [q1, q2], [u1, u2])
```

It is also important to supply the order of coordinates and speeds properly if there are dependent coordinates and speeds. They must be supplied after independent coordinates and speeds or as a keyword argument; this is shown later.

```
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> # Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> # We need the constraint equations to enter them though
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u4])
```

Additionally, if there are auxiliary speeds, they need to be identified here. See the examples for more information on this. In this example `u4` is the auxiliary speed.

```
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u2, u3], u_auxiliary=[u4])
```

Kinematic differential equations must also be supplied; there are to be provided as a list of expressions which are each equal to zero. A trivial example follows:

```
>>> kd = [q1d - u1, q2d - u2]
```

Turning on `mechanics_printing()` makes the expressions significantly shorter and is recommended. Alternatively, the `mprint` and `mpprint` commands can be used.

If there are non-holonomic constraints, dependent speeds need to be specified (and so do dependent coordinates, but they only come into play when linearizing the system). The constraints need to be supplied in a list of expressions which are equal to zero, trivial motion and configuration constraints are shown below:

```
>>> N = ReferenceFrame('N')
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> q1d, q2d, q3d, q4d = dynamicsymbols('q1 q2 q3 q4', 1)
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> #Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> speed_cons = [u2 - u1, u3 - u1 - u4]
>>> coord_cons = [q2 - q1]
```

(continues on next page)

(continued from previous page)

```

>>> q_ind = [q1, q3, q4]
>>> q_dep = [q2]
>>> u_ind = [u1, u4]
>>> u_dep = [u2, u3]
>>> kd = [q1d - u1, q2d - u2, q3d - u3, q4d - u4]
>>> KM = KanesMethod(N, q_ind, u_ind, kd,
...                 q_dependent=q_dep,
...                 configuration_constraints=coord_cons,
...                 u_dependent=u_dep,
...                 velocity_constraints=speed_cons)

```

A dictionary returning the solved $\dot{q}$'s can also be solved for:

```

>>> mechanics_printing(pretty_print=False)
>>> KM.kindiffdict()
{q1': u1, q2': u2, q3': u3, q4': u4}

```

The final step in forming the equations of motion is supplying a list of bodies and particles, and a list of 2-tuples of the form (Point, Vector) or (ReferenceFrame, Vector) to represent applied forces and torques.

```

>>> N = ReferenceFrame('N')
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> Pa = Particle('Pa', P, 5)
>>> BL = [Pa]
>>> FL = [(P, 7 * N.x)]
>>> KM = KanesMethod(N, [q], [u], [qd - u])
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> KM.mass_matrix
Matrix([[5]])
>>> KM.forcing
Matrix([[7]])

```

When there are motion constraints, the mass matrix is augmented by the $k_{dnh}(q, t)$ matrix, and the forcing vector by the $f_{dnh}(q, \dot{q}, u, t)$ vector.

There are also the “full” mass matrix and “full” forcing vector terms, these include the kinematic differential equations; the mass matrix is of size $(n + o) \times (n + o)$, or square and the size of all coordinates and speeds.

```

>>> KM.mass_matrix_full
Matrix([
[1, 0],
[0, 5]])
>>> KM.forcing_full
Matrix([
[u],
[7]])

```

Exploration of the provided examples is encouraged in order to gain more understanding of the KanesMethod object.

Lagrange's Method in Physics/Mechanics

mechanics provides functionality for deriving equations of motion using [Lagrange's method](#). This document will describe Lagrange's method as used in this module, but not how the equations are actually derived.

Structure of Equations

In mechanics we are assuming there are 3 basic sets of equations needed to describe a system; the constraint equations, the time differentiated constraint equations and the dynamic equations.

$$\begin{aligned}\mathbf{m}_c(q, t)\dot{q} + \mathbf{f}_c(q, t) &= 0 \\ \mathbf{m}_{dc}(\dot{q}, q, t)\ddot{q} + \mathbf{f}_{dc}(\dot{q}, q, t) &= 0 \\ \mathbf{m}_d(\dot{q}, q, t)\ddot{q} + \square_c(q, t)\lambda + \mathbf{f}_d(\dot{q}, q, t) &= 0\end{aligned}$$

In this module, the expressions formed by using Lagrange's equations of the second kind are rearranged into the following form:

$$\mathbf{M}(q, t)x = \mathbf{f}(q, \dot{q}, t)$$

where in the case of a system without constraints:

$$x = \ddot{q}$$

For a constrained system with n generalized speeds and m constraints, we will get $n - m$ equations. The mass-matrix/forcing equations are then augmented in the following fashion:

$$\begin{aligned}x &= \begin{bmatrix} \ddot{q} \\ \lambda \end{bmatrix} \\ \mathbf{M}(q, t) &= [\mathbf{m}_d(q, t) \quad \square_c(q, t)] \\ \mathbf{F}(\dot{q}, q, t) &= [\mathbf{f}_d(q, \dot{q}, t)]\end{aligned}$$

Lagrange's Method in Physics/Mechanics

The formulation of the equations of motion in mechanics using Lagrange's Method starts with the creation of generalized coordinates and a Lagrangian. The Lagrangian can either be created with the Lagrangian function or can be a user supplied function. In this case we will supply the Lagrangian.

```
>>> from sympy.physics.mechanics import *
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> L = q1d**2 + q2d**2
```

To formulate the equations of motion we create a `LagrangesMethod` object. The Lagrangian and generalized coordinates need to be supplied upon initialization.

```
>>> LM = LagrangesMethod(L, [q1, q2])
```

With that the equations of motion can be formed.

```
>>> mechanics_printing(pretty_print=False)
>>> LM.form_lagranges_equations()
Matrix([
[2*q1''],
[2*q2'']])
```

It is possible to obtain the mass matrix and the forcing vector.

```
>>> LM.mass_matrix
Matrix([
[2, 0],
[0, 2]])

>>> LM.forcing
Matrix([
[0],
[0]])
```

If there are any holonomic or non-holonomic constraints, they must be supplied as keyword arguments (`hol_coneqs` and `nonhol_coneqs` respectively) in a list of expressions which are equal to zero. Modifying the example above, the equations of motion can then be generated:

```
>>> LM = LagrangesMethod(L, [q1, q2], hol_coneqs=[q1 - q2])
```

When the equations of motion are generated in this case, the Lagrange multipliers are introduced; they are represented by lam1 in this case. In general, there will be as many multipliers as there are constraint equations.

```
>>> LM.form_lagranges_equations()
Matrix([
[ lam1 + 2*q1''],
[-lam1 + 2*q2'']])
```

Also in the case of systems with constraints, the ‘full’ mass matrix is augmented by the $k_{dc}(q, t)$ matrix, and the forcing vector by the $f_{dc}(q, \dot{q}, t)$ vector. The ‘full’ mass matrix is of size $(2n + o) \times (2n + o)$, i.e. it’s a square matrix.

```
>>> LM.mass_matrix_full
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 2, 0, -1],
[0, 0, 0, 2, 1],
[0, 0, 1, -1, 0]])
>>> LM.forcing_full
Matrix([
[q1'],
[q2'],
[ 0],
[ 0],
[ 0]])
```

If there are any non-conservative forces or moments acting on the system, they must also be supplied as keyword arguments in a list of 2-tuples of the form `(Point, Vector)` or `(ReferenceFrame, Vector)` where the `Vector` represents the non-conservative forces and torques. Along with this 2-tuple, the inertial frame must also be specified as a keyword argument. This is shown below by modifying the example above:

```
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, q1d * N.x)
>>> FL = [(P, 7 * N.x)]
>>> LM = LagrangesMethod(L, [q1, q2], forcelist=FL, frame=N)
>>> LM.form_lagranges_equations()
Matrix([
[2*q1'' - 7],
[ 2*q2'']])
```

Exploration of the provided examples is encouraged in order to gain more understanding of the `LagrangesMethod` object.

Symbolic Systems in Physics/Mechanics

The *SymbolicSystem* class in physics/mechanics is a location for the pertinent information of a multibody dynamic system. In its most basic form it contains the equations of motion for the dynamic system, however, it can also contain information regarding the loads that the system is subject to, the bodies that the system is comprised of and any additional equations the user feels is important for the system. The goal of this class is to provide a unified output format for the equations of motion that numerical analysis code can be designed around.

SymbolicSystem Example Usage

This code will go over the manual input of the equations of motion for the simple pendulum that uses the Cartesian location of the mass as the generalized coordinates into *SymbolicSystem*.

The equations of motion are formed in the physics/mechanics/examples. In that spot the variables `q1` and `q2` are used in place of `x` and `y` and the reference frame is rotated 90 degrees.

```
>>> from sympy import atan, symbols, Matrix
>>> from sympy.physics.mechanics import (dynamicsymbols, ReferenceFrame,
...                                     Particle, Point)
>>> import sympy.physics.mechanics.system as system
```

The first step will be to initialize all of the dynamic and constant symbols.

```
>>> x, y, u, v, lam = dynamicsymbols('x y u v lambda')
>>> m, l, g = symbols('m l g')
```

Next step is to define the equations of motion in multiple forms:

- [1] **Explicit form where the kinematics and dynamics are combined** $x' = F_1(x, t, r, p)$
- [2] **Implicit form where the kinematics and dynamics are combined** $M_2(x, p) x' = F_2(x, t, r, p)$
- [3] **Implicit form where the kinematics and dynamics are separate** $M_3(q, p) u' = F_3(q, u, t, r, p)$ $q' = G(q, u, t, r, p)$

where

`x` : states, e.g. `[q, u]` `t` : time `r` : specified (exogenous) inputs `p` : constants `q` : generalized coordinates `u` : generalized speeds `F_1` : right hand side of the combined equations in explicit form `F_2` : right hand side of the combined equations

in implicit form F_3 : right hand side of the dynamical equations in implicit form
 M_2 : mass matrix of the combined equations in implicit form M_3 : mass matrix
of the dynamical equations in implicit form G : right hand side of the kinematical
differential equations

```
>>> dyn_implicit_mat = Matrix([[1, 0, -x/m],
...                             [0, 1, -y/m],
...                             [0, 0, l**2/m]])
>>> dyn_implicit_rhs = Matrix([0, 0, u**2 + v**2 - g*y])
>>> comb_implicit_mat = Matrix([[1, 0, 0, 0, 0],
...                             [0, 1, 0, 0, 0],
...                             [0, 0, 1, 0, -x/m],
...                             [0, 0, 0, 1, -y/m],
...                             [0, 0, 0, 0, l**2/m]])
>>> comb_implicit_rhs = Matrix([u, v, 0, 0, u**2 + v**2 - g*y])
>>> kin_explicit_rhs = Matrix([u, v])
>>> comb_explicit_rhs = comb_implicit_mat.LUsolve(comb_implicit_rhs)
```

Now the reference frames, points and particles will be set up so this information can be passed into *system.SymbolicSystem* in the form of a bodies and loads iterable.

```
>>> theta = atan(x/y)
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [theta, N.z])
>>> A.set_ang_vel(N, omega * N.z)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> P = O.locatenew('P', l * A.x)
>>> P.v2pt_theory(O, N, A)
l*omega*A.y
>>> Pa = Particle('Pa', P, m)
```

Now the bodies and loads iterables need to be initialized.

```
>>> bodies = [Pa]
>>> loads = [(P, g * m * N.x)]
```

The equations of motion are in the form of a differential algebraic equation (DAE) and DAE solvers need to know which of the equations are the algebraic expressions. This information is passed into *SymbolicSystem* as a list specifying which rows are the algebraic equations. In this example it is a different row based on the chosen equations of motion format. The row index should always correspond to the mass matrix that is being input to the *SymbolicSystem* class but will always correspond to the row index of the combined dynamics and kinematics when being accessed from the *SymbolicSystem* class.

```
>>> alg_con = [2]
>>> alg_con_full = [4]
```

An iterable containing the states now needs to be created for the system. The *SymbolicSystem* class can determine which of the states are considered coordinates or speeds by passing in the indexes of the coordinates and speeds. If these indexes are not passed in the object will not be able to differentiate between coordinates and speeds.

```
>>> states = (x, y, u, v, lam)
>>> coord_idx = (0, 1)
>>> speed_idx = (2, 3)
```

Now the equations of motion instances can be created using the above mentioned equations of motion formats.

```
>>> symsystem1 = system.SymbolicSystem(states, comb_explicit_rhs,
...                                     alg_con=alg_con_full, bodies=bodies,
...                                     loads=loads)
>>> symsystem2 = system.SymbolicSystem(states, comb_implicit_rhs,
...                                     mass_matrix=comb_implicit_mat,
...                                     alg_con=alg_con_full,
...                                     coord_idx=coord_idx)
>>> symsystem3 = system.SymbolicSystem(states, dyn_implicit_rhs,
...                                     mass_matrix=dyn_implicit_mat,
...                                     coordinate_derivatives=kin_explicit_rhs,
...                                     alg_con=alg_con,
...                                     coord_idx=coord_idx,
...                                     speed_idx=speed_idx)
```

Like coordinates and speeds, the bodies and loads attributes can only be accessed if they are specified during initialization of the 'SymbolicSystem' class. Lastly here are some attributes accessible from the 'SymbolicSystem' class. ::

```
>>> symsystem1.states
Matrix([
[ x(t)],
[ y(t)],
[ u(t)],
[ v(t)],
[ lambda(t)]]])
>>> symsystem2.coordinates
Matrix([
[x(t)],
[y(t)]]])
>>> symsystem3.speeds
Matrix([
[u(t)],
[v(t)]]])
>>> symsystem1.comb_explicit_rhs
Matrix([
[ u(t)],
[ v(t)],
[ (-g*y(t) + u(t)**2 + v(t)**2)*x(t)/l**2],
[ (-g*y(t) + u(t)**2 + v(t)**2)*y(t)/l**2],
[ m*(-g*y(t) + u(t)**2 + v(t)**2)/l**2]])
>>> symsystem2.comb_implicit_rhs
Matrix([
[ u(t)],
[ v(t)],
[ 0],
[ 0],
[ -g*y(t) + u(t)**2 + v(t)**2]])
>>> symsystem2.comb_implicit_mat
Matrix([
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, -x(t)/m],
[0, 0, 0, 1, -y(t)/m],
[0, 0, 0, 0, l**2/m]])
```

(continues on next page)

(continued from previous page)

```

>>> symsystem3.dyn_implicit_rhs
Matrix([
[
0],
[
0],
[-g*y(t) + u(t)**2 + v(t)**2]])
>>> symsystem3.dyn_implicit_mat
Matrix([
[1, 0, -x(t)/m],
[0, 1, -y(t)/m],
[0, 0, l**2/m]])
>>> symsystem3.kin_explicit_rhs
Matrix([
[u(t)],
[v(t)]]
>>> symsystem1.alg_con
[4]
>>> symsystem1.bodies
(Pa,)
>>> symsystem1.loads
((P, g*m*N.x),)

```

Linearization in Physics/Mechanics

mechanics includes methods for linearizing the generated equations of motion (EOM) about an operating point (also known as the trim condition). Note that this operating point doesn't have to be an equilibrium position, it just needs to satisfy the equations of motion.

Linearization is accomplished by taking the first order Taylor expansion of the EOM about the operating point. When there are no dependent coordinates or speeds this is simply the jacobian of the right hand side about q and u . However, in the presence of constraints more care needs to be taken. The linearization methods provided here handle these constraints correctly.

Background

In mechanics we assume all systems can be represented in the following general form:

$$\begin{aligned}
 f_c(q, t) &= 0_{l \times 1} \\
 f_v(q, u, t) &= 0_{m \times 1} \\
 f_a(q, \dot{q}, u, \dot{u}, t) &= 0_{m \times 1} \\
 f_0(q, \dot{q}, t) + f_1(q, u, t) &= 0_{n \times 1} \\
 f_2(q, u, \dot{u}, t) + f_3(q, \dot{q}, u, r, t) + f_4(q, \lambda, t) &= 0_{(o-m+k) \times 1}
 \end{aligned}$$

where

$$\begin{aligned}
 q, \dot{q} &\in \mathbb{R}^n \\
 u, \dot{u} &\in \mathbb{R}^o \\
 r &\in \mathbb{R}^s \\
 \lambda &\in \mathbb{R}^k
 \end{aligned}$$

In this form,

- f_c represents the configuration constraint equations
- f_v represents the velocity constraint equations
- f_a represents the acceleration constraint equations
- f_0 and f_1 form the kinematic differential equations
- f_2 , f_3 , and f_4 form the dynamic differential equations
- q and $\dot{q}$ are the generalized coordinates and their derivatives
- u and $\dot{u}$ are the generalized speeds and their derivatives
- r is the system inputs
- λ is the Lagrange multipliers

This generalized form is held inside the `Linearizer` class, which performs the actual linearization. Both `KanesMethod` and `LagrangesMethod` objects have methods for forming the linearizer using the `to_linearizer` class method.

A Note on Dependent Coordinates and Speeds

If the system being linearized contains constraint equations, this results in not all generalized coordinates being independent (i.e. q_1 may depend on q_2). With l configuration constraints, and m velocity constraints, there are l dependent coordinates and m dependent speeds.

In general, you may pick any of the coordinates and speeds to be dependent, but in practice some choices may result in undesirable singularities. Methods for deciding which coordinates/speeds to make dependent is behind the scope of this guide. For more information, please see [Blajer1994] (page 1873).

Once the system is coerced into the generalized form, the linearized EOM can be solved for. The methods provided in `mechanics` allow for two different forms of the linearized EOM:

M , A , and B In this form, the forcing matrix is linearized into two separate matrices A and B . This is the default form of the linearized EOM. The resulting equations are:

$$M \begin{bmatrix} \delta \dot{q} \\ \delta \dot{u} \\ \delta \lambda \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \end{bmatrix} + B [\delta r]$$

where

$$\begin{aligned} M &\in \mathbb{R}^{(n+o+k) \times (n+o+k)} \\ A &\in \mathbb{R}^{(n+o+k) \times (n-l+o-m)} \\ B &\in \mathbb{R}^{(n+o+k) \times s} \end{aligned}$$

Note that q_i and u_i are just the independent coordinates and speeds, while q and u contains both the independent and dependent coordinates and speeds.

A and B In this form, the linearized EOM are brought into explicit first order form, in terms of just the independent coordinates and speeds. This form is often used in stability analysis or control theory. The resulting equations are:

$$\begin{bmatrix} \delta \dot{q}_i \\ \delta \dot{u}_i \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \end{bmatrix} + B [\delta r]$$

where

$$A \in \mathbb{R}^{(n-l+o-m) \times (n-l+o-m)}$$

$$B \in \mathbb{R}^{(n-l+o-m) \times s}$$

To use this form set `A_and_B=True` in the `linearize` class method.

Linearizing Kane's Equations

After initializing the `KanesMethod` object and forming F_r and F_r^* using the `kanes_equations` class method, linearization can be accomplished in a couple ways. The different methods will be demonstrated with a simple pendulum system:

```
>>> from sympy import symbols, Matrix
>>> from sympy.physics.mechanics import *
>>> q1 = dynamicsymbols('q1')           # Angle of pendulum
>>> u1 = dynamicsymbols('u1')           # Angular velocity
>>> q1d = dynamicsymbols('q1', 1)
>>> L, m, t, g = symbols('L, m, t, g')

>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, u1*N.z)

>>> # Locate point P relative to the origin N*
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Create Kinematic Differential Equations
>>> kde = Matrix([q1d - u1])

>>> # Input the force resultant at P
>>> R = m*g*N.x

>>> # Solve for eom with kanes method
>>> KM = KanesMethod(N, q_ind=[q1], u_ind=[u1], kd_eqs=kde)
>>> fr, frstar = KM.kanes_equations([pP], [(P, R)])
```

1. Using the Linearizer class directly:

A linearizer object can be created using the `to_linearizer` class method. This coerces the representation found in the `KanesMethod` object into the generalized form described above. As the independent and dependent coordinates and speeds are specified upon creation of the `KanesMethod` object, there is no need to specify them here.

```
>>> linearizer = KM.to_linearizer()
```

The linearized EOM can then be formed with the `linearize` method of the `Linearizer` object:

```
>>> M, A, B = linearizer.linearize()
>>> M
Matrix([
[1,      0],
[0, -L**2*m]])
>>> A
Matrix([
[      0, 1],
[L*g*m*cos(q1(t)), 0]])
>>> B
Matrix(0, 0, [])
```

Alternatively, the A and B form can be generated instead by specifying `A_and_B=True`:

```
>>> A, B = linearizer.linearize(A_and_B=True)
>>> A
Matrix([
[      0, 1],
[-g*cos(q1(t))/L, 0]])
>>> B
Matrix(0, 0, [])
```

An operating point can also be specified as a dictionary or an iterable of dictionaries. This will evaluate the linearized form at the specified point before returning the matrices:

```
>>> op_point = {q1: 0, u1: 0}
>>> A_op, B_op = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A_op
Matrix([
[      0, 1],
[-g/L, 0]])
```

Note that the same effect can be had by applying `msubs` to the matrices generated without the `op_point` kwarg:

```
>>> assert msubs(A, op_point) == A_op
```

Sometimes the returned matrices may not be in the most simplified form. Simplification can be performed after the fact, or the `Linearizer` object can be made to perform simplification internally by setting the `simplify` kwarg to `True`.

2. Using the `linearize` class method:

The `linearize` method of the `KanesMethod` class is provided as a nice wrapper that calls `to_linearizer` internally, performs the linearization, and returns the result. Note that all the kwargs available in the `linearize` method described above are also available here:

```
>>> A, B, inp_vec = KM.linearize(A_and_B=True, op_point=op_point, new_method=True)
>>> A
Matrix([
[      0, 1],
[-g/L, 0]])
```

The additional output `inp_vec` is a vector containing all found dynamicsymbols not included in the generalized coordinate or speed vectors. These are assumed to be inputs to the system,

forming the r vector described in the background above. In this example there are no inputs, so the vector is empty:

```
>>> inp_vec
Matrix(0, 0, [])
```

What's with the new_method kwarg?

Previous releases of SymPy contained a linearization method for *KanesMethod* objects. This method is deprecated, and will be removed from future releases. Until then, you must set `new_method=True` in all calls to `KanesMethod.linearize`. After the old method is removed, this kwarg will no longer be needed.

Linearizing Lagrange's Equations

Linearization of Lagrange's equations proceeds much the same as that of Kane's equations. As before, the process will be demonstrated with a simple pendulum system:

```
>>> # Redefine A and P in terms of q1d, not u1
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, q1d*N.z)
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Solve for eom with Lagrange's method
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1], forcelist=[(P, R)], frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```

1. Using the Linearizer class directly:

A `Linearizer` object can be formed from a `LagrangesMethod` object using the `to_linearizer` class method. The only difference between this process and that of the `KanesMethod` class is that the `LagrangesMethod` object doesn't already have its independent and dependent coordinates and speeds specified internally. These must be specified in the call to `to_linearizer`. In this example there are no dependent coordinates and speeds, but if there were they would be included in the `q_dep` and `qd_dep` kwargs:

```
>>> linearizer = LM.to_linearizer(q_ind=[q1], qd_ind=[q1d])
```

Once in this form, everything is the same as it was before with the `KanesMethod` example:

```
>>> A, B = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A
Matrix([
[ 0, 1],
[-g/L, 0]])
```

2. Using the linearize class method:

Similar to KanesMethod, the LagrangesMethod class also provides a linearize method as a nice wrapper that calls to_linearizer internally, performs the linearization, and returns the result. As before, the only difference is that the independent and dependent coordinates and speeds must be specified in the call as well:

```
>>> A, B, inp_vec = LM.linearize(q_ind=[q1], qd_ind=[q1d], A_and_B=True, op_point=op_
↪point)
>>> A
Matrix([
[      0, 1],
[-g/L, 0]])
```

Potential Issues

While the Linearizer class should be able to linearize all systems, there are some potential issues that could occur. These are discussed below, along with some troubleshooting tips for solving them.

1. Symbolic linearization with A_and_B=True is slow

This could be due to a number of things, but the most likely one is that solving a large linear system symbolically is an expensive operation. Specifying an operating point will reduce the expression size and speed this up. If a purely symbolic solution is desired though (for application of many operating points at a later period, for example) a way to get around this is to evaluate with A_and_B=False, and then solve manually after applying the operating point:

```
>>> M, A, B = linearizer.linearize()
>>> M_op = msubs(M, op_point)
>>> A_op = msubs(A, op_point)
>>> perm_mat = linearizer.perm_mat
>>> A_lin = perm_mat.T * M_op.LUsolve(A_op)
>>> A_lin
Matrix([
[      0, 1],
[-g/L, 0]])
```

The fewer symbols in A and M before solving, the faster this solution will be. Thus, for large expressions, it may be to your benefit to delay conversion to the A and B form until most symbols are subbed in for their numeric values.

2. The linearized form has nan, zoo, or oo as matrix elements

There are two potential causes for this. The first (and the one you should check first) is that some choices of dependent coordinates will result in singularities at certain operating points. Coordinate partitioning in a systemic manner to avoid this is beyond the scope of this guide; see [Blajer1994] (page 1873) for more information.

The other potential cause for this is that the matrices may not have been in the most reduced form before the operating point was substituted in. A simple example of this behavior is:


```
>>> from sympy import sin, tan
>>> expr = sin(q1)/tan(q1)
>>> op_point = {q1: 0}
>>> expr.subs(op_point)
nan
```

Note that if this expression was simplified before substitution, the correct value results:

```
>>> expr.simplify().subs(op_point)
1
```

A good way of avoiding this hasn't been found yet. For expressions of reasonable size, using `msubs` with `smart=True` will apply an algorithm that tries to avoid these conditions. For large expressions though this is extremely time consuming.

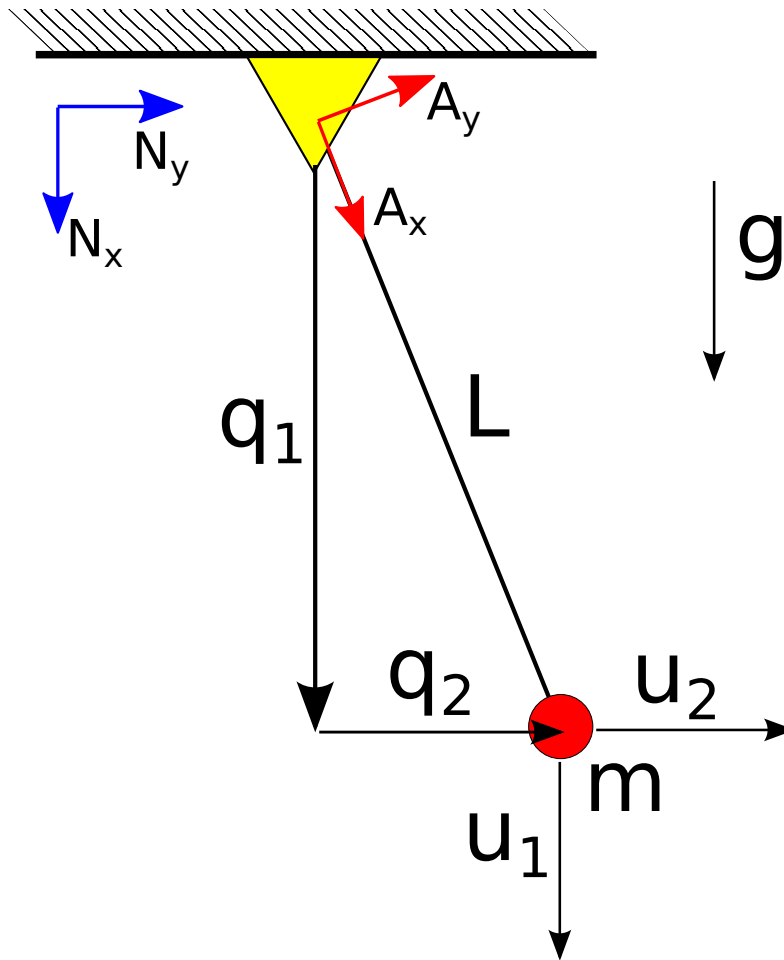
```
>>> msubs(expr, op_point, smart=True)
1
```

Further Examples

The pendulum example used above was simple, but didn't include any dependent coordinates or speeds. For a more thorough example, the same pendulum was linearized with dependent coordinates using both Kane's and Lagrange's methods:

Nonminimal Coordinates Pendulum

In this example we demonstrate the use of the functionality provided in `mechanics` for deriving the equations of motion (EOM) for a pendulum with a nonminimal set of coordinates. As the pendulum is a one degree of freedom system, it can be described using one coordinate and one speed (the pendulum angle, and the angular velocity respectively). Choosing instead to describe the system using the x and y coordinates of the mass results in a need for constraints. The system is shown below:



The system will be modeled using both Kane's and Lagrange's methods, and the resulting EOM linearized. While this is a simple problem, it should illustrate the use of the linearization methods in the presence of constraints.

Kane's Method

First we need to create the dynamicsymbols needed to describe the system as shown in the above diagram. In this case, the generalized coordinates q_1 and q_2 represent the mass x and y coordinates in the inertial N frame. Likewise, the generalized speeds u_1 and u_2 represent the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.

```
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix, solve
>>> # Create generalized coordinates and speeds for this non-minimal realization
>>> # q1, q2 = N.x and N.y coordinates of pendulum
>>> # u1, u2 = N.x and N.y velocities of pendulum
>>> q1, q2 = dynamicsymbols('q1:3')
>>> q1d, q2d = dynamicsymbols('q1:3', level=1)
>>> u1, u2 = dynamicsymbols('u1:3')
>>> u1d, u2d = dynamicsymbols('u1:3', level=1)
>>> L, m, g, t = symbols('L, m, g, t')
```

Next, we create a world coordinate frame N , and its origin point N^* . The velocity of the origin is set to 0. A second coordinate frame A is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

```
>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> theta1 = atan(q2/q1)
>>> A = N.orientnew('A', 'axis', [theta1, N.z])
```

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A Particle object is then created to represent the mass at this location.

```
>>> # Locate the pendulum mass
>>> P = pN.locatenew('P1', q1*N.x + q2*N.y)
>>> pP = Particle('pP', P, m)
```

The kinematic differential equations (KDEs) relate the derivatives of the generalized coordinates to the generalized speeds. In this case the speeds are the derivatives, so these are simple. A dictionary is also created to map $\dot{q}$ to u :

```
>>> # Calculate the kinematic differential equations
>>> kde = Matrix([q1d - u1,
...               q2d - u2])
>>> dq_dict = solve(kde, [q1d, q2d])
```

The velocity of the mass is then the time derivative of the position from the origin N^* :

```
>>> # Set velocity of point P
>>> P.set_vel(N, P.pos_from(pN).dt(N).subs(dq_dict))
```

As this system has more coordinates than degrees of freedom, constraints are needed. The configuration constraints relate the coordinates to each other. In this case the constraint is that the distance from the origin to the mass is always the length L (the pendulum doesn't get longer). Likewise, the velocity constraint is that the mass velocity in the $A.x$ direction is always 0 (no radial velocity).

```
>>> f_c = Matrix([P.pos_from(pN).magnitude() - L])
>>> f_v = Matrix([P.vel(N).express(A).dot(A.x)])
>>> f_v.simplify()
```

The force on the system is just gravity, at point P.

```
>>> # Input the force resultant at P
>>> R = m*g*N.x
```

With the problem setup, the equations of motion can be generated using the KanesMethod class. As there are constraints, dependent and independent coordinates need to be provided to the class. In this case we'll use q_2 and u_2 as the independent coordinates and speeds:

```
>>> # Derive the equations of motion using the KanesMethod class.
>>> KM = KanesMethod(N, q_ind=[q2], u_ind=[u2], q_dependent=[q1],
...                  u_dependent=[u1], configuration_constraints=f_c,
```

(continues on next page)

(continued from previous page)

```
... velocity_constraints=f_v, kd_eqs=kde)
>>> (fr, frstar) = KM.kanes_equations([pP],[P, R])
```

For linearization, operating points can be specified on the call, or be substituted in afterwards. In this case we'll provide them in the call, supplied in a list. The `A_and_B=True` kwarg indicates to solve invert the M matrix and solve for just the explicit linearized A and B matrices. The `simplify=True` kwarg indicates to simplify inside the linearize call, and return the presimplified matrices. The cost of doing this is small for simple systems, but for larger systems this can be a costly operation, and should be avoided.

```
>>> # Set the operating point to be straight down, and non-moving
>>> q_op = {q1: L, q2: 0}
>>> u_op = {u1: 0, u2: 0}
>>> ud_op = {uld: 0, u2d: 0}
>>> # Perform the linearization
>>> A, B, inp_vec = KM.linearize(op_point=[q_op, u_op, ud_op], A_and_B=True,
...                               new_method=True, simplify=True)
>>> A
Matrix([
[ 0, 1],
[-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting A matrix has dimensions 2×2 , while the number of total states is $\text{len}(q) + \text{len}(u) = 2 + 2 = 4$. This is because for constrained systems the resulting `A_and_B` form has a partitioned state vector only containing the independent coordinates and speeds. Written out mathematically, the system linearized about this point would be written as:

$$\begin{bmatrix} \dot{q}_2 \\ \dot{u}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{L} & 0 \end{bmatrix} \begin{bmatrix} q_2 \\ u_2 \end{bmatrix}$$

Lagrange's Method

The derivation using Lagrange's method is very similar to the approach using Kane's method described above. As before, we first create the dynamicsymbols needed to describe the system. In this case, the generalized coordinates q_1 and q_2 represent the mass x and y coordinates in the inertial N frame. This results in the time derivatives $\dot{q}_1$ and $\dot{q}_2$ representing the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.

```
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix
>>> q1, q2 = dynamicsymbols('q1:3')
>>> q1d, q2d = dynamicsymbols('q1:3', level=1)
>>> L, m, g, t = symbols('L, m, g, t')
```

Next, we create a world coordinate frame N , and its origin point N^* . The velocity of the origin is set to 0. A second coordinate frame A is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

```
>>> # Compose World Frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
```

(continues on next page)

(continued from previous page)

```
>>> pN.set_vel(N, 0)
>>> # A.x is along the pendulum
>>> theta1 = atan(q2/q1)
>>> A = N.orientnew('A', 'axis', [theta1, N.z])
```

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A Particle object is then created to represent the mass at this location.

```
>>> # Create point P, the pendulum mass
>>> P = pN.locatenew('P1', q1*N.x + q2*N.y)
>>> P.set_vel(N, P.pos_from(pN).dt(N))
>>> pP = Particle('pP', P, m)
```

As this system has more coordinates than degrees of freedom, constraints are needed. In this case only a single holonomic constraints is needed: the distance from the origin to the mass is always the length L (the pendulum doesn't get longer).

```
>>> # Holonomic Constraint Equations
>>> f_c = Matrix([q1**2 + q2**2 - L**2])
```

The force on the system is just gravity, at point P.

```
>>> # Input the force resultant at P
>>> R = m*g*N.x
```

With the problem setup, the Lagrangian can be calculated, and the equations of motion formed. Note that the call to `LagrangesMethod` includes the Lagrangian, the generalized coordinates, the constraints (specified by `hol_coneqs` or `nonhol_coneqs`), the list of (body, force) pairs, and the inertial frame. In contrast to the `KanesMethod` initializer, independent and dependent coordinates are not partitioned inside the `LagrangesMethod` object. Such a partition is supplied later.

```
>>> # Calculate the lagrangian, and form the equations of motion
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1, q2], hol_coneqs=f_c, forcelist=[(P, R)], frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```

Next, we compose the operating point dictionary, set in the hanging at rest position:

```
>>> # Compose operating point
>>> op_point = {q1: L, q2: 0, q1d: 0, q2d: 0, q1d.diff(t): 0, q2d.diff(t): 0}
```

As there are constraints in the formulation, there will be corresponding Lagrange Multipliers. These may appear inside the linearized form as well, and thus should also be included inside the operating point dictionary. Fortunately, the `LagrangesMethod` class provides an easy way of solving for the multipliers at a given operating point using the `solve_multipliers` method.

```
>>> # Solve for multiplier operating point
>>> lam_op = LM.solve_multipliers(op_point=op_point)
```

With this solution, linearization can be completed. Note that in contrast to the `KanesMethod` approach, the `LagrangesMethod.linearize` method also requires the partitioning of the generalized coordinates and their time derivatives into independent and dependent vectors. This is the same as what was passed into the `KanesMethod` constructor above:

```
>>> op_point.update(lam_op)
>>> # Perform the Linearization
>>> A, B, inp_vec = LM.linearize([q2], [q2d], [q1], [q1d],
...                             op_point=op_point, A_and_B=True)
>>> A
Matrix([
[ 0, 1],
[-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting A matrix has dimensions 2×2 , while the number of total states is $2 \times \text{len}(q) = 4$. This is because for constrained systems the resulting A_and_B form has a partitioned state vector only containing the independent coordinates and their derivatives. Written out mathematically, the system linearized about this point would be written as:

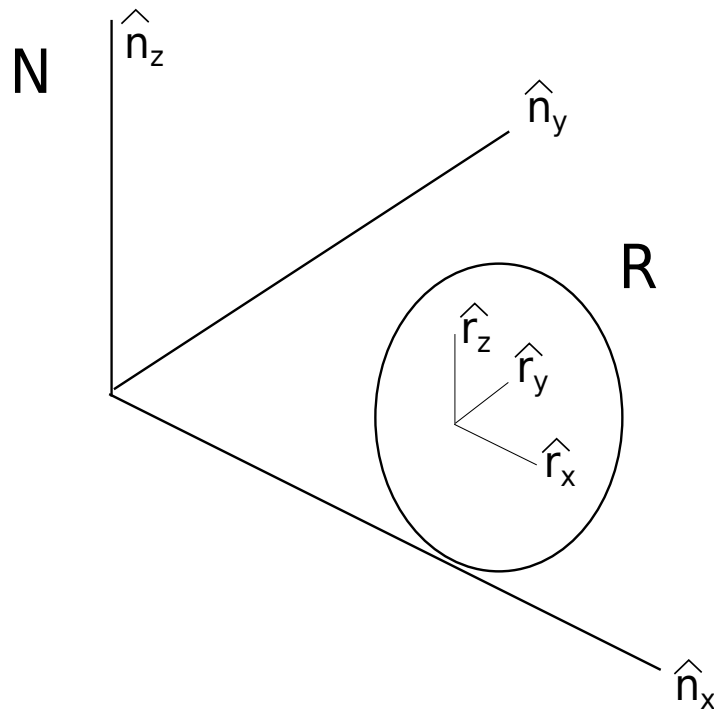
$$\begin{bmatrix} \dot{q}_2 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{L} & 0 \end{bmatrix} \begin{bmatrix} q_2 \\ \dot{q}_2 \end{bmatrix}$$

Examples for Physics/Mechanics

Here are some examples that illustrate how one typically uses this module. We have ordered the examples roughly according to increasing difficulty. If you have used this module to do something others might find useful or interesting, consider adding it here!

A rolling disc

The disc is assumed to be infinitely thin, in contact with the ground at only 1 point, and it is rolling without slip on the ground. See the image below.



We model the rolling disc in three different ways, to show more of the functionality of this module.

A rolling disc, with Kane's method

Here the definition of the rolling disc's kinematics is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and three speed variables are need to describe this system, along with the disc's mass and radius, and the local gravity (note that mass will drop out).

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
>>> mechanics_printing(pretty_print=False)
```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame's basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame's basis (the lean frame); it is for this reason that we defined intermediate frames, rather than using a body-three orientation.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc's center of mass. Finally we form the velocity and acceleration of the disc.

```
>>> C = Point('C')
>>> C.set_vel(N, 0)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> Dmc.v2pt_theory(C, N, R)
r*u2*L.x - r*u1*L.y
```

This is a simple way to form the inertia dyadic. The inertia of the disc does not change within the lean frame as the disc rolls; this will make for simpler equations in the end.

```
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
```

Kinematic differential equations; how the generalized coordinate time derivatives relate to generalized speeds.

```
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Creation of the force list; it is the gravitational force at the center of mass of the disc. Then we create the disc by assigning a Point to the center of mass attribute, a ReferenceFrame to the frame attribute, and mass and inertia. Then we form the body list.

```
>>> ForceList = [(Dmc, - m * g * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]
```

Finally we form the equations of motion, using the same steps we did before. Specify inertial frame, supply generalized coordinates and speeds, supply kinematic differential equation dictionary, compute F_r from the force list and F_r^* from the body list, compute the mass matrix and forcing terms, then solve for the u dots (time derivatives of the generalized speeds).

```
>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd)
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
>>> rhs.simplify()
>>> mprint(rhs)
Matrix([
[4*g*sin(q2) + 6*r*u2*u3 - r*u3**2*tan(q2))/(5*r)],
[
-2*u1*u3/3],
[
(-2*u2 + u3*tan(q2))*u1]])
```

A rolling disc, with Kane's method and constraint forces

We will now revisit the rolling disc example, except this time we are bringing the non-contributing (constraint) forces into evidence. See [Kane1985] (page 1873) for a more thorough explanation of this. Here, we will turn on the automatic simplification done when doing vector operations. It makes the outputs nicer for small problems, but can cause larger vector operations to hang.

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
```

These two lines introduce the extra quantities needed to find the constraint forces.

```
>>> u4, u5, u6, f1, f2, f3 = dynamicsymbols('u4 u5 u6 f1 f2 f3')
```

Most of the main code is the same as before.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

The definition of rolling without slip necessitates that the velocity of the contact point is zero; as part of bringing the constraint forces into evidence, we have to introduce speeds at this point, which will by definition always be zero. They are normal to the ground, along the path which the disc is rolling, and along the ground in a perpendicular direction.


```
>>> C = Point('C')
>>> C.set_vel(N, u4 * L.x + u5 * (Y.z ^ L.x) + u6 * Y.z)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> vel = Dmc.v2pt_theory(C, N, R)
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Just as we previously introduced three speeds as part of this process, we also introduce three forces; they are in the same direction as the speeds, and represent the constraint forces in those directions.

```
>>> ForceList = [(Dmc, - m * g * Y.z), (C, f1 * L.x + f2 * (Y.z ^ L.x) + f3 * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]

>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd,
...                 u_auxiliary=[u4, u5, u6])
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
>>> rhs.simplify()
>>> mprint(rhs)
Matrix([
[(4*g*sin(q2) + 6*r*u2*u3 - r*u3**2*tan(q2))/(5*r)],
[
-2*u1*u3/3],
[
(-2*u2 + u3*tan(q2))*u1]])
>>> from sympy import trigsimp, signsimp, collect, factor_terms
>>> def simplify_auxiliary_eqs(w):
...     return signsimp(trigsimp(collect(collect(factor_terms(w), f2), m*r)))
>>> mprint(KM.auxiliary_eqs.applyfunc(simplify_auxiliary_eqs))
Matrix([
[
-m*r*(u1*u3 + u2') + f1],
[-m*r*u1**2*sin(q2) - m*r*u2*u3/cos(q2) + m*r*cos(q2)*u1' + f2],
[
-g*m + m*r*(u1**2*cos(q2) + sin(q2)*u1') + f3]])
```

A rolling disc using Lagrange's Method

Here the rolling disc is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and 3 speed variables are needed to describe this system, along with the disc's mass and radius, and the local gravity.

```
>>> from sympy import symbols, cos, sin
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> q1d, q2d, q3d = dynamicsymbols('q1 q2 q3', 1)
>>> r, m, g = symbols('r m g')
```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame's basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame's basis (the lean frame).

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
```

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc's center of mass. Finally we form the velocity and acceleration of the disc.

```
>>> C = Point('C')
>>> C.set_vel(N, 0)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> Dmc.v2pt_theory(C, N, R)
r*(sin(q2)*q1' + q3')*L.x - r*q2'*L.y
```

Forming the inertia dyadic.

```
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
```

We then set the potential energy and determine the Lagrangian of the rolling disc.

```
>>> BodyD.potential_energy = - m * g * r * cos(q2)
>>> Lag = Lagrangian(N, BodyD)
```

Then the equations of motion are generated by initializing the `LagrangesMethod` object. Finally we solve for the generalized accelerations(q double dots) with the `rhs` method.

```
>>> q = [q1, q2, q3]
>>> l = LagrangesMethod(Lag, q)
>>> le = l.form_lagranges_equations()
>>> le = le.simplify()
>>> le
Matrix([
[m*r**2*(6*sin(q2)*q3'' + 5*sin(2*q2)*q1'*q2' + 6*cos(q2)*q2'*q3' - 5*cos(2*q2)*q1''/
↪ 2 + 7*q1''/2)/4],
[
↪ m*r*(4*g*sin(q2) - 5*r*sin(2*q2)*q1''**2/2 - 6*r*cos(q2)*q1'*q3
↪ ' + 5*r*q2'')/4],
[
↪ 3*m*r**2*(sin(q2)*q1'' + cos(q2)*q1
↪ '*q2' + q3'')/2]])
>>> lrhs = l.rhs().simplify()
>>> lrhs
Matrix([
[
↪ q1'],
[
↪ q2'],
[
↪ q3'],
[
↪ -2*(2*tan(q2)*q1' + 3*q3'/cos(q2))*q2'],
[-4*g*sin(q2)/(5*r) + sin(2*q2)*q1''**2/2 + 6*cos(q2)*q1'*q3'/5],
[
↪ (-5*cos(q2)*q1' + 6*tan(q2)*q3' + 4*q1'/cos(q2))*q2']]])
```

A bicycle

The bicycle is an interesting system in that it has multiple rigid bodies, non-holonomic constraints, and a holonomic constraint. The linearized equations of motion are presented in

[Meijaard2007] (page 1873). This example will go through construction of the equations of motion in mechanics.

```
>>> from sympy import *
>>> from sympy.physics.mechanics import *
>>> print('Calculation of Linearized Bicycle \'A\' Matrix, '
...       'with States: Roll, Steer, Roll Rate, Steer Rate')
Calculation of Linearized Bicycle "A" Matrix, with States: Roll, Steer, Roll Rate,
↳Steer Rate
```

Note that this code has been crudely ported from Autolev, which is the reason for some of the unusual naming conventions. It was purposefully as similar as possible in order to aid initial porting & debugging. We set `Vector.simp` to `False` (in case it has been set `True` elsewhere), since it slows down the computations:

```
>>> Vector.simp = False
>>> mechanics_printing(pretty_print=False)
```

Declaration of Coordinates & Speeds: A simple definition for `qdots`, `qd = u`, is used in this code. Speeds are: yaw frame ang. rate, roll frame ang. rate, rear wheel frame ang. rate (spinning motion), frame ang. rate (pitching motion), steering frame ang. rate, and front wheel ang. rate (spinning motion). Wheel positions are ignorable coordinates, so they are not introduced.

```
>>> q1, q2, q3, q4, q5 = dynamicsymbols('q1 q2 q3 q4 q5')
>>> q1d, q2d, q4d, q5d = dynamicsymbols('q1 q2 q4 q5', 1)
>>> u1, u2, u3, u4, u5, u6 = dynamicsymbols('u1 u2 u3 u4 u5 u6')
>>> u1d, u2d, u3d, u4d, u5d, u6d = dynamicsymbols('u1 u2 u3 u4 u5 u6', 1)
```

Declaration of System's Parameters: The below symbols should be fairly self-explanatory.

```
>>> WFrads, WRrads, htangle, forkoffset = symbols('WFrads WRrads htangle forkoffset')
>>> forklength, framelength, forkcg1 = symbols('forklength framelength forkcg1')
>>> forkcg3, framecg1, framecg3, Iwr11 = symbols('forkcg3 framecg1 framecg3 Iwr11')
>>> Iwr22, Iwf11, Iwf22, Iframe11 = symbols('Iwr22 Iwf11 Iwf22 Iframe11')
>>> Iframe22, Iframe33, Iframe31, Ifork11 = \
...     symbols('Iframe22 Iframe33 Iframe31 Ifork11')
>>> Ifork22, Ifork33, Ifork31, g = symbols('Ifork22 Ifork33 Ifork31 g')
>>> mframe, mfork, mwf, mwr = symbols('mframe mfork mwf mwr')
```

Set up reference frames for the system: N - inertial Y - yaw R - roll WR - rear wheel, rotation angle is ignorable coordinate so not oriented Frame - bicycle frame TempFrame - statically rotated frame for easier reference inertia definition Fork - bicycle fork TempFork - statically rotated frame for easier reference inertia definition WF - front wheel, again possesses an ignorable coordinate

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> R = Y.orientnew('R', 'Axis', [q2, Y.x])
>>> Frame = R.orientnew('Frame', 'Axis', [q4 + htangle, R.y])
>>> WR = ReferenceFrame('WR')
>>> TempFrame = Frame.orientnew('TempFrame', 'Axis', [-htangle, Frame.y])
>>> Fork = Frame.orientnew('Fork', 'Axis', [q5, Frame.x])
>>> TempFork = Fork.orientnew('TempFork', 'Axis', [-htangle, Fork.y])
>>> WF = ReferenceFrame('WF')
```

Kinematics of the Bicycle: First block of code is forming the positions of the relevant points rear wheel contact -> rear wheel's center of mass -> frame's center of mass + frame/fork

connection -> fork's center of mass + front wheel's center of mass -> front wheel contact point.

```
>>> WR_cont = Point('WR_cont')
>>> WR_mc = WR_cont.locatenew('WR_mc', WRrad * R.z)
>>> Steer = WR_mc.locatenew('Steer', framelength * Frame.z)
>>> Frame_mc = WR_mc.locatenew('Frame_mc', -framecg1 * Frame.x + framecg3 * Frame.z)
>>> Fork_mc = Steer.locatenew('Fork_mc', -forkcg1 * Fork.x + forkcg3 * Fork.z)
>>> WF_mc = Steer.locatenew('WF_mc', forklength * Fork.x + forkoffset * Fork.z)
>>> WF_cont = WF_mc.locatenew('WF_cont', WFrads*(dot(Fork.y, Y.z)*Fork.y - \
...                               Y.z).normalize())
```

Set the angular velocity of each frame: Angular accelerations end up being calculated automatically by differentiating the angular velocities when first needed. :: u1 is yaw rate u2 is roll rate u3 is rear wheel rate u4 is frame pitch rate u5 is fork steer rate u6 is front wheel rate

```
>>> Y.set_ang_vel(N, u1 * Y.z)
>>> R.set_ang_vel(Y, u2 * R.x)
>>> WR.set_ang_vel(Frame, u3 * Frame.y)
>>> Frame.set_ang_vel(R, u4 * Frame.y)
>>> Fork.set_ang_vel(Frame, u5 * Fork.x)
>>> WF.set_ang_vel(Fork, u6 * Fork.y)
```

Form the velocities of the points, using the 2-point theorem. Accelerations again are calculated automatically when first needed.

```
>>> WR_cont.set_vel(N, 0)
>>> WR_mc.v2pt_theory(WR_cont, N, WR)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y
>>> Steer.v2pt_theory(WR_mc, N, Frame)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) + u4)*Frame.
↪ x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4))*Frame.y
>>> Frame_mc.v2pt_theory(WR_mc, N, Frame)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framecg3*(u1*sin(q2) + u4)*Frame.x
↪ + (-framecg1*(u1*cos(htangle + q4)*cos(q2) + u2*sin(htangle + q4)) - framecg3*(-
↪ u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4)))*Frame.y +
↪ framecg1*(u1*sin(q2) + u4)*Frame.z
>>> Fork_mc.v2pt_theory(Steer, N, Fork)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) + u4)*Frame.
↪ x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4))*Frame.y +
↪ forkcg3*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 + u2*sin(htangle
↪ + q4)*sin(q5) + u4*cos(q5))*Fork.x + (-forkcg1*(-sin(q2)*sin(q5) + cos(htangle +
↪ q4)*cos(q2)*cos(q5))*u1 + u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkcg3*(-
↪ u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y +
↪ forkcg1*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 + u2*sin(htangle
↪ + q4)*sin(q5) + u4*cos(q5))*Fork.z
>>> WF_mc.v2pt_theory(Steer, N, Fork)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) + u4)*Frame.
↪ x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4))*Frame.y +
↪ forkoffset*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +
↪ u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.x + (forklength*(-sin(q2)*sin(q5)
↪ + cos(htangle + q4)*cos(q2)*cos(q5))*u1 + u2*sin(htangle + q4)*cos(q5) -
↪ u4*sin(q5)) - forkoffset*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4) +
↪ u5))*Fork.y - forklength*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1
↪ + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z
>>> WF_cont.v2pt_theory(WF_mc, N, WF)
- WFrads*(-sin(q2)*sin(q5)*cos(htangle + q4) + cos(q2)*cos(q5))*u6 + u4*cos(q2) +
↪ u5*sin(htangle + q4)*sin(q2))/sqrt((-sin(q2)*cos(q5) - sin(q5)*cos(htangle + q4)*cos(q2))
↪ + 1)*Y.x +
↪ WFrads*(u2 + u5*cos(htangle + q4) + u6*sin(htangle + q4)*sin(q5))/sqrt((-
↪ sin(q2)*cos(q5) - sin(q5)*cos(htangle + q4)*cos(q2)) + 1)*Y.y + WRrad*(u1*sin(q2) + u3 + u4)*R.x -
↪ WRrad*u2*R.y + framelength*(u1*sin(q2) + u4)*Frame.x - framelength*(-u1*sin(htangle
↪ + q4)*cos(q2) + u2*cos(htangle + q4))*Frame.y + (-WFrads*(sin(q2)*cos(q5) +
↪ sin(q5)*cos(htangle + q4)*cos(q2))*((-sin(q2)*sin(q5) + cos(htangle +
↪ q4)*cos(q2)*cos(q5)))*u1 + u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkcg3*(-
↪ u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y - forklength*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1
↪ + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z
```

(continued from previous page)

Sets the inertias of each body. Uses the inertia frame to construct the inertia dyadics. Wheel inertias are only defined by principal moments of inertia, and are in fact constant in the frame and fork reference frames; it is for this reason that the orientations of the wheels does not need to be defined. The frame and fork inertias are defined in the 'Temp' frames which are fixed to the appropriate body frames; this is to allow easier input of the reference values of the benchmark paper. Note that due to slightly different orientations, the products of inertia need to have their signs flipped; this is done later when entering the numerical value.

```
>>> Frame_I = (inertia(TempFrame, Iframe11, Iframe22, Iframe33, 0, 0,
...                               Iframe31), Frame_mc)
>>> Fork_I = (inertia(TempFork, Ifork11, Ifork22, Ifork33, 0, 0, Ifork31), Fork_mc)
>>> WR_I = (inertia(Frame, Iwr11, Iwr22, Iwr11), WR_mc)
>>> WF_I = (inertia(Fork, Iwf11, Iwf22, Iwf11), WF_mc)
```

Declaration of the RigidBody containers.

```
>>> BodyFrame = RigidBody('BodyFrame', Frame_mc, Frame, mframe, Frame_I)
>>> BodyFork = RigidBody('BodyFork', Fork_mc, Fork, mfork, Fork_I)
>>> BodyWR = RigidBody('BodyWR', WR_mc, WR, mwr, WR_I)
>>> BodyWF = RigidBody('BodyWF', WF_mc, WF, mwf, WF_I)

>>> print('Before Forming the List of Nonholonomic Constraints.')
Before Forming the List of Nonholonomic Constraints.
```

The kinematic differential equations; they are defined quite simply. Each entry in this list is equal to zero.

```
>>> kd = [q1d - u1, q2d - u2, q4d - u4, q5d - u5]
```

The nonholonomic constraints are the velocity of the front wheel contact point dotted into the X, Y, and Z directions; the yaw frame is used as it is "closer" to the front wheel (1 less DCM connecting them). These constraints force the velocity of the front wheel contact point to be 0 in the inertial frame; the X and Y direction constraints enforce a "no-slip" condition, and the Z direction constraint forces the front wheel contact point to not move away from the ground frame, essentially replicating the holonomic constraint which does not allow the frame pitch to change in an invalid fashion.

```
>>> conlist_speed = [WF_cont.vel(N) & Y.x,
...                  WF_cont.vel(N) & Y.y,
...                  WF_cont.vel(N) & Y.z]
```

The holonomic constraint is that the position from the rear wheel contact point to the front wheel contact point when dotted into the normal-to-ground plane direction must be zero; effectively that the front and rear wheel contact points are always touching the ground plane. This is actually not part of the dynamic equations, but instead is necessary for the linearization process.

```
>>> conlist_coord = [WF_cont.pos_from(WR_cont) & Y.z]
```

The force list; each body has the appropriate gravitational force applied at its center of mass.

```
>>> FL = [(Frame_mc, -mframe * g * Y.z), (Fork_mc, -mfork * g * Y.z),
...       (WF_mc, -mwf * g * Y.z), (WR_mc, -mwr * g * Y.z)]
>>> BL = [BodyFrame, BodyFork, BodyWR, BodyWF]
```

The N frame is the inertial frame, coordinates are supplied in the order of independent, dependent coordinates. The kinematic differential equations are also entered here. Here the independent speeds are specified, followed by the dependent speeds, along with the non-holonomic constraints. The dependent coordinate is also provided, with the holonomic constraint. Again, this is only comes into play in the linearization process, but is necessary for the linearization to correctly work.

```
>>> KM = KanesMethod(N, q_ind=[q1, q2, q5],
...                 q_dependent=[q4], configuration_constraints=conlist_coord,
...                 u_ind=[u2, u3, u5],
...                 u_dependent=[u1, u4, u6], velocity_constraints=conlist_speed,
...                 kd_eqs=kd)
>>> print('Before Forming Generalized Active and Inertia Forces, Fr and Fr*')
Before Forming Generalized Active and Inertia Forces, Fr and Fr*
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> print('Base Equations of Motion Computed')
Base Equations of Motion Computed
```

This is the start of entering in the numerical values from the benchmark paper to validate the eigenvalues of the linearized equations from this model to the reference eigenvalues. Look at the aforementioned paper for more information. Some of these are intermediate values, used to transform values from the paper into the coordinate systems used in this model.

```
>>> PaperRadRear = 0.3
>>> PaperRadFront = 0.35
>>> HTA = evalf.N(pi/2-pi/10)
>>> TrailPaper = 0.08
>>> rake = evalf.N(-(TrailPaper*sin(HTA) - (PaperRadFront*cos(HTA))))
>>> PaperWb = 1.02
>>> PaperFrameCgX = 0.3
>>> PaperFrameCgZ = 0.9
>>> PaperForkCgX = 0.9
>>> PaperForkCgZ = 0.7
>>> FrameLength = evalf.N(PaperWb*sin(HTA) - (rake - \
...                         (PaperRadFront - PaperRadRear)*cos(HTA)))
>>> FrameCGNorm = evalf.N((PaperFrameCgZ - PaperRadRear - \
...                         (PaperFrameCgX/sin(HTA))*cos(HTA))*sin(HTA))
>>> FrameCGPar = evalf.N((PaperFrameCgX / sin(HTA) + \
...                         (PaperFrameCgZ - PaperRadRear - \
...                         PaperFrameCgX / sin(HTA) * cos(HTA)) * cos(HTA)))
>>> tempa = evalf.N((PaperForkCgZ - PaperRadFront))
>>> tempb = evalf.N((PaperWb-PaperForkCgX))
>>> tempc = evalf.N(sqrt(tempa**2 + tempb**2))
>>> PaperForkL = evalf.N((PaperWb*cos(HTA) - \
...                         (PaperRadFront - PaperRadRear)*sin(HTA)))
>>> ForkCGNorm = evalf.N(rake + (tempc * sin(pi/2 - \
...                         HTA - acos(tempa/tempc))))
>>> ForkCGPar = evalf.N(tempc * cos((pi/2 - HTA) - \
...                         acos(tempa/tempc)) - PaperForkL)
```

Here is the final assembly of the numerical values. The symbol 'v' is the forward speed of the bicycle (a concept which only makes sense in the upright, static equilibrium case?). These are in a dictionary which will later be substituted in. Again the sign on the product of inertia values is flipped here, due to different orientations of coordinate systems.

```
>>> v = Symbol('v')
>>> val_dict = {
```

(continues on next page)

(continued from previous page)

```

...     WFrads: PaperRadFront,
...     WRads: PaperRadRear,
...     htangle: HTA,
...     forkoffset: rake,
...     forklenght: PaperForkL,
...     framelength: FrameLength,
...     forkcg1: ForkCGPar,
...     forkcg3: ForkCGNorm,
...     framecg1: FrameCGNorm,
...     framecg3: FrameCGPar,
...     Iwr11: 0.0603,
...     Iwr22: 0.12,
...     Iwf11: 0.1405,
...     Iwf22: 0.28,
...     Ifork11: 0.05892,
...     Ifork22: 0.06,
...     Ifork33: 0.00708,
...     Ifork31: 0.00756,
...     Iframe11: 9.2,
...     Iframe22: 11,
...     Iframe33: 2.8,
...     Iframe31: -2.4,
...     mfork: 4,
...     mframe: 85,
...     mwf: 3,
...     mwr: 2,
...     g: 9.81,
...     q1: 0,
...     q2: 0,
...     q4: 0,
...     q5: 0,
...     u1: 0,
...     u2: 0,
...     u3: v/PaperRadRear,
...     u4: 0,
...     u5: 0,
...     u6: v/PaperRadFront}
>>> kdd = KM.kindiffdict()
>>> print('Before Linearization of the \"Forcing\" Term')
Before Linearization of the \"Forcing\" Term

```

Linearizes the forcing vector; the equations are set up as $MM \dot{u} = \text{forcing}$, where MM is the mass matrix, $\dot{u}$ is the vector representing the time derivatives of the generalized speeds, and forcing is a vector which contains both external forcing terms and internal forcing terms, such as centripetal or Coriolis forces. This actually returns a matrix with as many rows as total coordinates and speeds, but only as many columns as independent coordinates and speeds. (Note that below this is commented out, as it takes a few minutes to run, which is not good when performing the doctests)

```
>>> # forcing_lin = KM.linearize()[0].subs(sub_dict)
```

As mentioned above, the size of the linearized forcing terms is expanded to include both q 's and u 's, so the mass matrix must have this done as well. This will likely be changed to be part of the linearized process, for future reference.


```
>>> MM_full = (KM._k_kqdot).row_join(zeros(4, 6)).col_join(
...     (zeros(6, 4)).row_join(KM.mass_matrix))
>>> print('Before Substitution of Numerical Values')
Before Substitution of Numerical Values
```

I think this is pretty self explanatory. It takes a really long time though. I've experimented with using evalf with substitution, this failed due to maximum recursion depth being exceeded; I also tried lambdifying this, and it is also not successful. (again commented out due to speed)

```
>>> # MM_full = MM_full.subs(val_dict)
>>> # forcing_lin = forcing_lin.subs(val_dict)
>>> # print('Before .evalf() call')

>>> # MM_full = MM_full.evalf()
>>> # forcing_lin = forcing_lin.evalf()
```

Finally, we construct an “A” matrix for the form $\dot{x} = A x$ (x being the state vector, although in this case, the sizes are a little off). The following line extracts only the minimum entries required for eigenvalue analysis, which correspond to rows and columns for lean, steer, lean rate, and steer rate. (this is all commented out due to being dependent on the above code, which is also commented out):

```
>>> # Amat = MM_full.inv() * forcing_lin
>>> # A = Amat.extract([1,2,4,6],[1,2,3,5])
>>> # print(A)
>>> # print('v = 1')
>>> # print(A.subs(v, 1).eigenvals())
>>> # print('v = 2')
>>> # print(A.subs(v, 2).eigenvals())
>>> # print('v = 3')
>>> # print(A.subs(v, 3).eigenvals())
>>> # print('v = 4')
>>> # print(A.subs(v, 4).eigenvals())
>>> # print('v = 5')
>>> # print(A.subs(v, 5).eigenvals())
```

Upon running the above code yourself, enabling the commented out lines, compare the computed eigenvalues to those in the referenced paper. This concludes the bicycle example.

Potential Issues/Advanced Topics/Future Features in Physics/Mechanics

This document will describe some of the more advanced functionality that this module offers but which is not part of the “official” interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.

Common Issues

Here issues with numerically integrating code, choice of dynamicsymbols for coordinate and speed representation, printing, differentiating, and substitution will occur.

Numerically Integrating Code

See Future Features: Code Output

Differentiating

Differentiation of very large expressions can take some time in SymPy; it is possible for large expressions to take minutes for the derivative to be evaluated. This will most commonly come up in linearization.

Choice of Coordinates and Speeds

The Kane object is set up with the assumption that the generalized speeds are not the same symbol as the time derivatives of the generalized coordinates. This isn't to say that they can't be the same, just that they have to have a different symbol. If you did this:

```
>> KM.coords([q1, q2, q3])
>> KM.speeds([q1d, q2d, q3d])
```

Your code would not work. Currently, kinematic differential equations are required to be provided. It is at this point that we hope the user will discover they should not attempt the behavior shown in the code above.

This behavior might not be true for other methods of forming the equations of motion though.

Printing

The default printing options are to use sorting for Vector and Dyad measure numbers, and have unsorted output from the `mprint`, `mpprint`, and `mlatex` functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

Substitution

There are two common issues with substitution in mechanics:

- When subbing in expressions for `dynamicsymbols`, sympy's normal subs will substitute in for derivatives of the dynamic symbol as well:

```
>>> from sympy.physics.mechanics import dynamicsymbols
>>> x = dynamicsymbols('x')
>>> expr = x.diff() + x
>>> sub_dict = {x: 1}
>>> expr.subs(sub_dict)
Derivative(1, t) + 1
```

In this case, `x` was replaced with 1 inside the `Derivative` as well, which is undesired.

- Substitution into large expressions can be slow.

If your substitution is simple (direct replacement of expressions with other expressions, such as when evaluating at an operating point) it is recommended to use the provided `msubs` function, as it is significantly faster, and handles the derivative issue appropriately:

```
>>> from sympy.physics.mechanics import msubs
>>> msubs(expr, sub_dict)
Derivative(x(t), t) + 1
```

Linearization

Currently, the linearization methods don't support cases where there are non-coordinate, non-speed dynamic symbols outside of the "dynamic equations". It also does not support cases where time derivatives of these types of dynamic symbols show up. This means if you have kinematic differential equations which have a non-coordinate, non-speed dynamic symbol, it will not work. It also means if you have defined a system parameter (say a length or distance or mass) as a dynamic symbol, its time derivative is likely to show up in the dynamic equations, and this will prevent linearization.

Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorter expressions at this point, which will result in shorter expressions later (such as when forming Kane's equations).

Advanced Interfaces

Advanced Functionality

Remember that the Kane object supports bodies which have time-varying masses and inertias, although this functionality isn't completely compatible with the linearization method.

Operators were discussed earlier as a potential way to do mathematical operations on Vector and Dyad objects. The majority of the code in this module is actually coded with them, as it can (subjectively) result in cleaner, shorter, more readable code. If using this interface in your code, remember to take care and use parentheses; the default order of operations in Python results in addition occurring before some of the vector products, so use parentheses liberally.

Future Features

This will cover the planned features to be added to this submodule.

Code Output

A function for generating code output for numerical integration is the highest priority feature to implement next. There are a number of considerations here.

Code output for C (using the GSL libraries), Fortran 90 (using LSODA), MATLAB, and SciPy is the goal. Things to be considered include: use of cse on large expressions for MATLAB and SciPy, which are interpretive. It is currently unclear whether compiled languages will benefit

from common subexpression elimination, especially considering that it is a common part of compiler optimization, and there can be a significant time penalty when calling `cse`.

Care needs to be taken when constructing the strings for these expressions, as well as handling of input parameters, and other dynamic symbols. How to deal with output quantities when integrating also needs to be decided, with the potential for multiple options being considered.

References for Physics/Mechanics

Mechanics API

Masses, Inertias & Particles, RigidBody (Docstrings)

Particle

class `sympy.physics.mechanics.particle.Particle(name, point, mass)`

A particle.

Particles have a non-zero mass and lack spatial extension; they take up no space.

Values need to be supplied on initialization, but can be changed later.

Parameters `name` : str

Name of particle

point : Point

A physics/mechanics Point which represents the position, velocity, and acceleration of this Particle

mass : sympifyable

A SymPy expression representing the Particle's mass

Examples

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> po = Point('po')
>>> m = Symbol('m')
>>> pa = Particle('pa', po, m)
>>> # Or you could change these later
>>> pa.mass = m
>>> pa.point = po
```

angular_momentum(point, frame)

Angular momentum of the particle about the point.

The angular momentum H , about some point O of a particle, P , is given by:

$$H = \mathbf{r} \times m * \mathbf{v}$$

where $\mathbf{r}$ is the position vector from point O to the particle P , m is the mass of the particle, and $\mathbf{v}$ is the velocity of the particle in the inertial frame, N .

Parameters `point` : Point

The point about which angular momentum of the particle is desired.

frame : ReferenceFrame

The frame in which angular momentum is desired.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point, ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> m, v, r = dynamicsymbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> A = O.locatenew('A', r * N.x)
>>> P = Particle('P', A, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.angular_momentum(O, N)
m*r*v*N.z
```

kinetic_energy(frame)

Kinetic energy of the particle

The kinetic energy, T , of a particle, P , is given by

$$T = \frac{1}{2} m v^2$$

where m is the mass of particle P , and v is the velocity of the particle in the supplied ReferenceFrame.

Parameters **frame** : ReferenceFrame

The Particle's velocity is typically defined with respect to an inertial frame but any relevant frame in which the velocity is known can be supplied.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point, ReferenceFrame
>>> from sympy import symbols
>>> m, v, r = symbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.kinetic_energy(N)
m*v**2/2
```

linear_momentum(frame)

Linear momentum of the particle.

The linear momentum L , of a particle P , with respect to frame N is given by

$$L = m * v$$

where m is the mass of the particle, and v is the velocity of the particle in the frame N .

Parameters **frame** : ReferenceFrame

The frame in which linear momentum is desired.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point, ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> m, v = dynamicsymbols('m v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> A = Particle('A', P, m)
>>> P.set_vel(N, v * N.x)
>>> A.linear_momentum(N)
m*v*N.x
```

mass

Mass of the particle.

point

Point of the particle.

potential_energy

The potential energy of the Particle.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import symbols
>>> m, g, h = symbols('m g h')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.potential_energy = m * g * h
>>> P.potential_energy
g*h*m
```

RigidBody

class sympy.physics.mechanics.rigidbody.**RigidBody**(name, masscenter, frame, mass, inertia)

An idealized rigid body.

This is essentially a container which holds the various components which describe a rigid body: a name, mass, center of mass, reference frame, and inertia.

All of these need to be supplied on creation, but can be changed afterwards.

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> from sympy.physics.mechanics import outer
>>> m = Symbol('m')
>>> A = ReferenceFrame('A')
>>> P = Point('P')
>>> I = outer(A.x, A.x)
>>> inertia_tuple = (I, P)
```

(continues on next page)

(continued from previous page)

```
>>> B = RigidBody('B', P, A, m, inertia_tuple)
>>> # Or you could change them afterwards
>>> m2 = Symbol('m2')
>>> B.mass = m2
```

Attributes

name	(string) The body's name.
masscenter	(Point) The point which represents the center of mass of the rigid body.
frame	(ReferenceFrame) The ReferenceFrame which the rigid body is fixed in.
mass	(Sympifyable) The body's mass.
inertia	((Dyadic, Point)) The body's inertia about a point; stored in a tuple as shown above.

angular_momentum(point, frame)

Returns the angular momentum of the rigid body about a point in the given frame.

The angular momentum H of a rigid body B about some point O in a frame N is given by:

$$H = I \cdot \omega + r \times Mv$$

where I is the central inertia dyadic of B , ω is the angular velocity of body B in the frame, N , r is the position vector from point O to the mass center of B , and v is the velocity of the mass center in the frame, N .

Parameters **point** : Point

The point about which angular momentum is desired.

frame : ReferenceFrame

The frame in which angular momentum is desired.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> M, v, r, omega = dynamicsymbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, 1 * N.x)
>>> I = outer(b.x, b.x)
>>> B = RigidBody('B', P, b, M, (I, P))
>>> B.angular_momentum(P, N)
omega*b.x
```

central_inertia

The body's central inertia dyadic.

kinetic_energy(frame)

Kinetic energy of the rigid body

The kinetic energy, T , of a rigid body, B , is given by

$$T = 1/2 (I \omega^2 + m v^2)$$

where I and m are the central inertia dyadic and mass of rigid body B , respectively, ω is the body's angular velocity and v is the velocity of the body's mass center in the supplied ReferenceFrame.

Parameters frame : ReferenceFrame

The RigidBody's angular velocity and the velocity of it's mass center are typically defined with respect to an inertial frame but any relevant frame in which the velocities are known can be supplied.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody
>>> from sympy import symbols
>>> M, v, r, omega = symbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer(b.x, b.x)
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, inertia_tuple)
>>> B.kinetic_energy(N)
M*v**2/2 + omega**2/2
```

linear_momentum(frame)

Linear momentum of the rigid body.

The linear momentum L , of a rigid body B , with respect to frame N is given by

$$L = M * v^*$$

where M is the mass of the rigid body and v^* is the velocity of the mass center of B in the frame, N .

Parameters frame : ReferenceFrame

The frame in which linear momentum is desired.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> M, v = dynamicsymbols('M v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer(N.x, N.x)
>>> Inertia_tuple = (I, P)
```

(continues on next page)

(continued from previous page)

```
>>> B = RigidBody('B', P, N, M, Inertia_tuple)
>>> B.linear_momentum(N)
M*v*N.x
```

potential_energy

The potential energy of the RigidBody.

Examples

```
>>> from sympy.physics.mechanics import RigidBody, Point, outer, \
↳ ReferenceFrame
>>> from sympy import symbols
>>> M, g, h = symbols('M g h')
>>> b = ReferenceFrame('b')
>>> P = Point('P')
>>> I = outer(b.x, b.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, Inertia_tuple)
>>> B.potential_energy = M * g * h
>>> B.potential_energy
M*g*h
```

inertia

`sympy.physics.mechanics.functions.inertia(frame, ixx, iyy, izz, ixy=0, iyz=0, izx=0)`

Simple way to create inertia Dyadic object.

If you don't know what a Dyadic is, just treat this like the inertia tensor. Then, do the easy thing and define it in a body-fixed frame.

Parameters **frame** : ReferenceFrame

The frame the inertia is defined in

ixx : Sympifyable

the xx element in the inertia dyadic

iyy : Sympifyable

the yy element in the inertia dyadic

izz : Sympifyable

the zz element in the inertia dyadic

ixy : Sympifyable

the xy element in the inertia dyadic

iyz : Sympifyable

the yz element in the inertia dyadic

izx : Sympifyable

the zx element in the inertia dyadic

Examples

```
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
```

inertia_of_point_mass

`sympy.physics.mechanics.functions.inertia_of_point_mass(mass, pos_vec, frame)`

Inertia dyadic of a point mass relative to point O.

Parameters **mass** : Sympifyable

Mass of the point mass

pos_vec : Vector

Position from point O to point mass

frame : ReferenceFrame

Reference frame to express the dyadic in

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import ReferenceFrame, inertia_of_point_mass
>>> N = ReferenceFrame('N')
>>> r, m = symbols('r m')
>>> px = r * N.x
>>> inertia_of_point_mass(m, px, N)
m*r**2*(N.y|N.y) + m*r**2*(N.z|N.z)
```

linear_momentum

`sympy.physics.mechanics.functions.linear_momentum(frame, *body)`

Linear momentum of the system.

This function returns the linear momentum of a system of Particle's and/or RigidBody's. The linear momentum of a system is equal to the vector sum of the linear momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The linear momentum of the system, L, is equal to the vector sum of the linear momentum of the particle, L1, and the linear momentum of the rigid body, L2, i.e.

$$L = L1 + L2$$

Parameters **frame** : ReferenceFrame

The frame in which linear momentum is desired.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose linear momentum is required.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, linear_momentum
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = Point('Ac')
>>> Ac.set_vel(N, 25 * N.y)
>>> I = outer(N.x, N.x)
>>> A = RigidBody('A', Ac, N, 20, (I, Ac))
>>> linear_momentum(N, A, Pa)
10*N.x + 500*N.y
```

angular_momentum

`sympy.physics.mechanics.functions.angular_momentum(point, frame, *body)`

Angular momentum of a system

This function returns the angular momentum of a system of Particle's and/or RigidBody's. The angular momentum of such a system is equal to the vector sum of the angular momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The angular momentum of the system, H, is equal to the vector sum of the angular momentum of the particle, H1, and the angular momentum of the rigid body, H2, i.e.

$$H = H1 + H2$$

Parameters `point` : Point

The point about which angular momentum of the system is desired.

frame : ReferenceFrame

The frame in which angular momentum is desired.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose angular momentum is required.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, angular_momentum
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
```

(continues on next page)

(continued from previous page)

```
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> angular_momentum(0, N, Pa, A)
10*N.z
```

kinetic_energy

`sympy.physics.mechanics.functions.kinetic_energy(frame, *body)`

Kinetic energy of a multibody system.

This function returns the kinetic energy of a system of Particle's and/or RigidBody's. The kinetic energy of such a system is equal to the sum of the kinetic energies of its constituents. Consider a system, S, comprising a rigid body, A, and a particle, P. The kinetic energy of the system, T, is equal to the vector sum of the kinetic energy of the particle, T1, and the kinetic energy of the rigid body, T2, i.e.

$$T = T1 + T2$$

Kinetic energy is a scalar.

Parameters **frame** : ReferenceFrame

The frame in which the velocity or angular velocity of the body is defined.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose kinetic energy is required.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, kinetic_energy
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> kinetic_energy(N, Pa, A)
350
```

potential_energy

`sympy.physics.mechanics.functions.potential_energy(*body)`

Potential energy of a multibody system.

This function returns the potential energy of a system of Particle's and/or RigidBody's. The potential energy of such a system is equal to the sum of the potential energy of its

constituents. Consider a system, S , comprising a rigid body, A , and a particle, P . The potential energy of the system, V , is equal to the vector sum of the potential energy of the particle, V_1 , and the potential energy of the rigid body, V_2 , i.e.

$$V = V_1 + V_2$$

Potential energy is a scalar.

Parameters *body1, body2, body3...* : Particle and/or RigidBody

The body (or bodies) whose potential energy is required.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, potential_energy
>>> from sympy import symbols
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> Pa = Particle('Pa', P, m)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> a = ReferenceFrame('a')
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
>>> potential_energy(Pa, A)
M*g*h + g*h*m
```

Lagrangian

`sympy.physics.mechanics.functions.Lagrangian(frame, *body)`

Lagrangian of a multibody system.

This function returns the Lagrangian of a system of Particle's and/or RigidBody's. The Lagrangian of such a system is equal to the difference between the kinetic energies and potential energies of its constituents. If T and V are the kinetic and potential energies of a system then it's Lagrangian, L , is defined as

$$L = T - V$$

The Lagrangian is a scalar.

Parameters *frame* : ReferenceFrame

The frame in which the velocity or angular velocity of the body is defined to determine the kinetic energy.

body1, body2, body3... : Particle and/or RigidBody

The body (or bodies) whose Lagrangian is required.

Examples

```
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, Lagrangian
>>> from sympy import symbols
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
>>> Lagrangian(N, Pa, A)
-M*g*h - g*h*m + 350
```

Kane's Method & Lagrange's Method (Docstrings)

KaneMethod

```
class sympy.physics.mechanics.kane.KanesMethod(frame, q_ind, u_ind,
                                                kd_eqs=None,
                                                q_dependent=None, config-
                                                uration_constraints=None,
                                                u_dependent=None, veloc-
                                                ity_constraints=None, accel-
                                                eration_constraints=None,
                                                u_auxiliary=None)
```

Kane's method object.

This object is used to do the “book-keeping” as you go through and form equations of motion in the way Kane presents in: Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

The attributes are for equations in the form $[M] \dot{u} = \text{forcing}$.

Examples

This is a simple example for a one degree of freedom translational spring-mass-damper. In this example, we first need to do the kinematics. This involves creating generalized speeds and coordinates and their derivatives. Then we create a point and set its velocity in a frame.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
```

(continues on next page)

(continued from previous page)

```

>>> from sympy.physics.mechanics import Point, Particle, KanesMethod
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> m, c, k = symbols('m c k')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)

```

Next we need to arrange/store information in the way that KanesMethod requires. The kinematic differential equations need to be stored in a dict. A list of forces/torques must be constructed, where each entry in the list is a (Point, Vector) or (ReferenceFrame, Vector) tuple, where the Vectors represent the Force or Torque. Next a particle needs to be created, and it needs to have a point and mass assigned to it. Finally, a list of all bodies and particles needs to be created.

```

>>> kd = [qd - u]
>>> FL = [(P, (-k * q - c * u) * N.x)]
>>> pa = Particle('pa', P, m)
>>> BL = [pa]

```

Finally we can generate the equations of motion. First we create the KanesMethod object and supply an inertial frame, coordinates, generalized speeds, and the kinematic differential equations. Additional quantities such as configuration and motion constraints, dependent coordinates and speeds, and auxiliary speeds are also supplied here (see the online documentation). Next we form FR^* and FR to complete: $Fr + Fr^* = 0$. We have the equations of motion at this point. It makes sense to rearrange them though, so we calculate the mass matrix and the forcing terms, for E.o.M. in the form: $[MM] \dot{u} = \text{forcing}$, where MM is the mass matrix, $\dot{u}$ is a vector of the time derivatives of the generalized speeds, and forcing is a vector representing “forcing” terms.

```

>>> KM = KanesMethod(N, q_ind=[q], u_ind=[u], kd_eqs=kd)
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> rhs
Matrix([[(-c*u(t) - k*q(t))/m]])
>>> KM.linearize(A_and_B=True)[0]
Matrix([
[ 0, 1],
[-k/m, -c/m]])

```

Please look at the documentation pages for more information on how to perform linearization and how to deal with dependent coordinates & speeds, and how do deal with bringing non-contributing forces into evidence.

Attributes

<code>q, u</code>	(Matrix) Matrices of the generalized coordinates and speeds
<code>bodylist</code>	(iterable) Iterable of Point and RigidBody objects in the system.
<code>forcelist</code>	(iterable) Iterable of (Point, vector) or (ReferenceFrame, vector) tuples describing the forces on the system.
<code>auxiliary</code>	(Matrix) If applicable, the set of auxiliary Kane's equations used to solve for non-contributing forces.
<code>mass_matrix</code>	(Matrix) The system's mass matrix
<code>forcing</code>	(Matrix) The system's forcing vector
<code>mass_matrix_full</code>	(Matrix) The "mass matrix" for the u's and q's
<code>forcing_full</code>	(Matrix) The "forcing vector" for the u's and q's

`auxiliary_eqs`

A matrix containing the auxiliary equations.

`forcing`

The forcing vector of the system.

`forcing_full`

The forcing vector of the system, augmented by the kinematic differential equations.

`kane's_equations(bodies, loads=None)`

Method to form Kane's equations, $Fr + Fr^* = 0$.

Returns (Fr, Fr^*) . In the case where auxiliary generalized speeds are present (say, s auxiliary speeds, o generalized speeds, and m motion constraints) the length of the returned vectors will be $o - m + s$ in length. The first $o - m$ equations will be the constrained Kane's equations, then the s auxiliary Kane's equations. These auxiliary equations can be accessed with the `auxiliary_eqs()`.

Parameters `bodies` : iterable

An iterable of all RigidBody's and Particle's in the system. A system must have at least one body.

loads : iterable

Takes in an iterable of (Particle, Vector) or (ReferenceFrame, Vector) tuples which represent the force at a point or torque on a frame. Must be either a non-empty iterable of tuples or None which corresponds to a system with no constraints.

`kindiffdict()`

Returns a dictionary mapping q' to u .

`linearize(**kwargs)`

Linearize the equations of motion about a symbolic operating point.

If kwarg `A_and_B` is False (default), returns M, A, B, r for the linearized form, $M[q', u']^T = A[q_{ind}, u_{ind}]^T + B*r$.

If kwarg `A_and_B` is True, returns A, B, r for the linearized form $dx = A*x + B*r$, where $x = [q_{ind}, u_{ind}]^T$. Note that this is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default `A_and_B=False`, returning M, A , and B . Values may then be substituted in to these matrices, and the state space form found as $A = P.T*M.inv()*A$, $B = P.T*M.inv()*B$, where $P = \text{Linearizer.perm\_mat}$.

In both cases, r is found as all dynamicsymbols in the equations of motion that are not part of q , u , q' , or u' . They are sorted in canonical form.

The operating points may be also entered using the `op_point` kwarg. This takes a dictionary of {symbol: value}, or a an iterable of such dictionaries. The values may be numeric or symbolic. The more values you can specify beforehand, the faster this computation will run.

For more documentation, please see the `Linearizer` class.

mass_matrix

The mass matrix of the system.

mass_matrix_full

The mass matrix of the system, augmented by the kinematic differential equations.

rhs(inv_method=None)

Returns the system's equations of motion in first order form. The output is the right hand side of:

$$\mathbf{x}' = \begin{bmatrix} q' \\ u' \end{bmatrix} =: \mathbf{f}(q, u, r, p, t)$$

The right hand side is what is needed by most numerical ODE integrators.

Parameters `inv_method` : str

The specific sympy inverse matrix calculation method to use. For a list of valid methods, see [inv\(\)](#) (page 746)

to_linearizer()

Returns an instance of the `Linearizer` class, initiated from the data in the `KanesMethod` class. This may be more desirable than using the `linearize` class method, as the `Linearizer` object will allow more efficient recalculation (i.e. about varying operating points).

LagrangesMethod

```
class sympy.physics.mechanics.lagrange.LagrangesMethod(Lagrangian,      qs,
                                                         forcelist=None,
                                                         bodies=None,
                                                         frame=None,
                                                         hol_coneqs=None,
                                                         non-
                                                         hol_coneqs=None)
```

Lagrange's method object.

This object generates the equations of motion in a two step procedure. The first step involves the initialization of `LagrangesMethod` by supplying the Lagrangian and the generalized coordinates, at the bare minimum. If there are any constraint equations, they can be supplied as keyword arguments. The Lagrange multipliers are automatically generated and are equal in number to the constraint equations. Similarly any non-conservative forces can be supplied in an iterable (as described below and also shown in the example) along with a `ReferenceFrame`. This is also discussed further in the `__init__` method.

Examples

This is a simple example for a one degree of freedom translational spring-mass-damper.

In this example, we first need to do the kinematics. This involves creating generalized coordinates and their derivatives. Then we create a point and set its velocity in a frame.

```
>>> from sympy.physics.mechanics import LagrangesMethod, Lagrangian
>>> from sympy.physics.mechanics import ReferenceFrame, Particle, Point
>>> from sympy.physics.mechanics import dynamicsymbols, kinetic_energy
>>> from sympy import symbols
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> m, k, b = symbols('m k b')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, qd * N.x)
```

We need to then prepare the information as required by LagrangesMethod to generate equations of motion. First we create the Particle, which has a point attached to it. Following this the lagrangian is created from the kinetic and potential energies. Then, an iterable of nonconservative forces/torques must be constructed, where each item is a (Point, Vector) or (ReferenceFrame, Vector) tuple, with the Vectors representing the nonconservative forces or torques.

```
>>> Pa = Particle('Pa', P, m)
>>> Pa.potential_energy = k * q**2 / 2.0
>>> L = Lagrangian(N, Pa)
>>> fl = [(P, -b * qd * N.x)]
```

Finally we can generate the equations of motion. First we create the LagrangesMethod object. To do this one must supply the Lagrangian, and the generalized coordinates. The constraint equations, the forcelist, and the inertial frame may also be provided, if relevant. Next we generate Lagrange's equations of motion, such that: Lagrange's equations of motion = 0. We have the equations of motion at this point.

```
>>> l = LagrangesMethod(L, [q], forcelist = fl, frame = N)
>>> print(l.form_lagranges_equations())
Matrix([[b*Derivative(q(t), t) + 1.0*k*q(t) + m*Derivative(q(t), (t, 2))]])
```

We can also solve for the states using the 'rhs' method.

```
>>> print(l.rhs())
Matrix([[Derivative(q(t), t)], [(-b*Derivative(q(t), t) - 1.0*k*q(t))/m]])
```

Please refer to the docstrings on each method for more details.

Attributes

<code>q, u</code>	(Matrix) Matrices of the generalized coordinates and speeds
<code>forcelist</code>	(iterable) Iterable of (Point, vector) or (ReferenceFrame, vector) tuples describing the forces on the system.
<code>bodies</code>	(iterable) Iterable containing the rigid bodies and particles of the system.
<code>mass_matrix</code>	(Matrix) The system's mass matrix
<code>forcing</code>	(Matrix) The system's forcing vector
<code>mass_matrix_full</code>	(Matrix) The "mass matrix" for the <code>qdot</code> 's, <code>qdoubledot</code> 's, and the lagrange multipliers (<code>lam</code>)
<code>forcing_full</code>	(Matrix) The forcing vector for the <code>qdot</code> 's, <code>qdoubledot</code> 's and lagrange multipliers (<code>lam</code>)

forcing

Returns the forcing vector from 'lagranges_equations' method.

forcing_full

Augments `qdots` to the forcing vector above.

form_lagranges_equations()

Method to form Lagrange's equations of motion.

Returns a vector of equations of motion using Lagrange's equations of the second kind.

linearize(q_ind=None, qd_ind=None, q_dep=None, qd_dep=None, **kwargs)

Linearize the equations of motion about a symbolic operating point.

If kwarg `A_and_B` is False (default), returns `M`, `A`, `B`, `r` for the linearized form, $M[q', u']^T = A[q_{ind}, u_{ind}]^T + B*r$.

If kwarg `A_and_B` is True, returns `A`, `B`, `r` for the linearized form $dx = A*x + B*r$, where $x = [q_{ind}, u_{ind}]^T$. Note that this is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default `A_and_B=False`, returning `M`, `A`, and `B`. Values may then be substituted in to these matrices, and the state space form found as $A = P.T*M.inv()*A$, $B = P.T*M.inv()*B$, where $P = \text{Linearizer.perm\_mat}$.

In both cases, `r` is found as all dynamicsymbols in the equations of motion that are not part of `q`, `u`, `q'`, or `u'`. They are sorted in canonical form.

The operating points may be also entered using the `op_point` kwarg. This takes a dictionary of {symbol: value}, or a an iterable of such dictionaries. The values may be numeric or symbolic. The more values you can specify beforehand, the faster this computation will run.

For more documentation, please see the `Linearizer` class.

mass_matrix

Returns the mass matrix, which is augmented by the Lagrange multipliers, if necessary.

If the system is described by '`n`' generalized coordinates and there are no constraint equations then an `n X n` matrix is returned.

If there are '`n`' generalized coordinates and '`m`' constraint equations have been supplied during initialization then an `n X (n+m)` matrix is returned. The $(n + m - 1)$ th and $(n + m)$ th columns contain the coefficients of the Lagrange multipliers.

mass_matrix_full

Augments the coefficients of qdots to the mass_matrix.

rhs(inv_method=None, **kwargs)

Returns equations that can be solved numerically

Parameters inv_method : str

The specific sympy inverse matrix calculation method to use. For a list of valid methods, see [inv\(\)](#) (page 746)

solve_multipliers(op_point=None, sol_type='dict')

Solves for the values of the lagrange multipliers symbolically at the specified operating point

Parameters op_point : dict or iterable of dicts, optional

Point at which to solve at. The operating point is specified as a dictionary or iterable of dictionaries of {symbol: value}. The value may be numeric or symbolic itself.

sol_type : str, optional

Solution return type. Valid options are: - 'dict': A dict of {symbol : value} (default) - 'Matrix': An ordered column matrix of the solution

to_linearizer(q_ind=None, qd_ind=None, q_dep=None, qd_dep=None)

Returns an instance of the Linearizer class, initiated from the data in the Lagranges-Method class. This may be more desirable than using the linearize class method, as the Linearizer object will allow more efficient recalculation (i.e. about varying operating points).

Parameters q_ind, qd_ind : array_like, optional

The independent generalized coordinates and speeds.

q_dep, qd_dep : array_like, optional

The dependent generalized coordinates and speeds.

SymbolicSystem (Docstrings)**SymbolicSystem**

```
class sympy.physics.mechanics.system.SymbolicSystem(coord_states,
                                                    right_hand_side,
                                                    speeds=None,
                                                    mass_matrix=None, coord-
                                                    nate_derivatives=None,
                                                    alg_con=None,      out-
                                                    put_eqns={},      co-
                                                    ord_idxs=None,
                                                    speed_idxs=None, bod-
                                                    ies=None, loads=None)
```

SymbolicSystem is a class that contains all the information about a system in a symbolic format such as the equations of motions and the bodies and loads in the system.

There are three ways that the equations of motion can be described for Symbolic System:

[1] Explicit form where the kinematics and dynamics are combined $\mathbf{x}'$
 $= F_1(\mathbf{x}, t, \mathbf{r}, \mathbf{p})$

[2] Implicit form where the kinematics and dynamics are combined

$$M\_2(x, p) x' = F\_2(x, t, r, p)$$

[3] Implicit form where the kinematics and dynamics are separate

$$M\_3(q, p) u' = F\_3(q, u, t, r, p) \quad q' = G(q, u, t, r, p)$$

where

x : states, e.g. $[q, u]$ t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds F_1 : right hand side of the combined equations in explicit form F_2 : right hand side of the combined equations in implicit form F_3 : right hand side of the dynamical equations in implicit form M_2 : mass matrix of the combined equations in implicit form M_3 : mass matrix of the dynamical equations in implicit form G : right hand side of the kinematical differential equations

Parameters **coord_states** : ordered iterable of functions of time

This input will either be a collection of the coordinates or states of the system depending on whether or not the speeds are also given. If speeds are specified this input will be assumed to be the coordinates otherwise this input will be assumed to be the states.

right_hand_side [Matrix] This variable is the right hand side of the equations of motion in any of the forms. The specific form will be assumed depending on whether a mass matrix or coordinate derivatives are given.

speeds [ordered iterable of functions of time, optional] This is a collection of the generalized speeds of the system. If given it will be assumed that the first argument (**coord_states**) will represent the generalized coordinates of the system.

mass_matrix [Matrix, optional] The matrix of the implicit forms of the equations of motion (forms [2] and [3]). The distinction between the forms is determined by whether or not the coordinate derivatives are passed in. If they are given form [3] will be assumed otherwise form [2] is assumed.

coordinate_derivatives [Matrix, optional] The right hand side of the kinematical equations in explicit form. If given it will be assumed that the equations of motion are being entered in form [3].

alg_con [Iterable, optional] The indexes of the rows in the equations of motion that contain algebraic constraints instead of differential equations. If the equations are input in form [3], it will be assumed the indexes are referencing the **mass_matrix/right_hand_side** combination and not the **coordinate_derivatives**.

output_eqns [Dictionary, optional] Any output equations that are desired to be tracked are stored in a dictionary where the key corresponds to the name given for the specific equation and the value is the equation itself in symbolic form

coord_idxs [Iterable, optional] If **coord_states** corresponds to the states rather than the coordinates this variable will tell SymbolicSystem which indexes of the states correspond to generalized coordinates.

speed_idxs [Iterable, optional] If **coord_states** corresponds to the states rather than the coordinates this variable will tell SymbolicSystem which indexes of the states correspond to generalized

speeds.

bodies [iterable of Body/Rigidbody objects, optional] Iterable containing the bodies of the system

loads [iterable of load instances (described below), optional] Iterable containing the loads of the system where forces are given by (point of application, force vector) and torques are given by (reference frame acting upon, torque vector). Ex [(point, force), (ref_frame, torque)]

Notes

m : number of generalized speeds n : number of generalized coordinates o : number of states

Example

As a simple example, the dynamics of a simple pendulum will be input into a SymbolicSystem object manually. First some imports will be needed and then symbols will be set up for the length of the pendulum (l), mass at the end of the pendulum (m), and a constant for gravity (g).

```
>>> from sympy import Matrix, sin, symbols
>>> from sympy.physics.mechanics import dynamicsymbols, SymbolicSystem
>>> l, m, g = symbols('l m g')
```

The system will be defined by an angle of θ from the vertical and a generalized speed of ω will be used where $\omega = \dot{\theta}$.

```
>>> theta, omega = dynamicsymbols('theta omega')
```

Now the equations of motion are ready to be formed and passed to the SymbolicSystem object.

```
>>> kin_explicit_rhs = Matrix([omega])
>>> dyn_implicit_mat = Matrix([l**2 * m])
>>> dyn_implicit_rhs = Matrix([-g * l * m * sin(theta)])
>>> symsystem = SymbolicSystem([theta], dyn_implicit_rhs, [omega],
...                             dyn_implicit_mat)
```

Attributes

co- or- di- nates	(Matrix, shape(n, 1)) This is a matrix containing the generalized coordinates of the system
speeds	(Matrix, shape(m, 1)) This is a matrix containing the generalized speeds of the system
states	(Matrix, shape(o, 1)) This is a matrix containing the state variables of the system
alg_con	(List) This list contains the indices of the algebraic constraints in the combined equations of motion. The presence of these constraints requires that a DAE solver be used instead of an ODE solver. If the system is given in form [3] the alg_con variable will be adjusted such that it is a representation of the combined kinematics and dynamics thus make sure it always matches the mass matrix entered.
dyn_in	(Matrix, shape(m, m)) This is the M matrix in form [3] of the equations of motion (the mass matrix or generalized inertia matrix of the dynamical equations of motion in implicit form).
dyn_in	(Matrix, shape(m, 1)) This is the F vector in form [3] of the equations of motion (the right hand side of the dynamical equations of motion in implicit form).
comb	(Matrix, shape(o, o)) This is the M matrix in form [2] of the equations of motion. This matrix contains a block diagonal structure where the top left block (the first rows) represent the matrix in the implicit form of the kinematical equations and the bottom right block (the last rows) represent the matrix in the implicit form of the dynamical equations.
comb	(Matrix, shape(o, 1)) This is the F vector in form [2] of the equations of motion. The top part of the vector represents the right hand side of the implicit form of the kinematical equations and the bottom of the vector represents the right hand side of the implicit form of the dynamical equations of motion.
comb	(Matrix, shape(o, 1)) This vector represents the right hand side of the combined equations of motion in explicit form (form [1] from above).
kin_exp	(Matrix, shape(m, 1)) This is the right hand side of the explicit form of the kinematical equations of motion as can be seen in form [3] (the G matrix).
out- put_eqs	(Dictionary) If output equations were given they are stored in a dictionary where the key corresponds to the name given for the specific equation and the value is the equation itself in symbolic form
bod- ies	(Tuple) If the bodies in the system were given they are stored in a tuple for future access
loads	(Tuple) If the loads in the system were given they are stored in a tuple for future access. This includes forces and torques where forces are given by (point of application, force vector) and torques are given by (reference frame acted upon, torque vector).

alg_con

Returns a list with the indices of the rows containing algebraic constraints in the combined form of the equations of motion

bodies

Returns the bodies in the system

comb_explicit_rhs

Returns the right hand side of the equations of motion in explicit form, $x' = F$, where the kinematical equations are included

comb_implicit_mat

Returns the matrix, M , corresponding to the equations of motion in implicit form (form [2]), $M \dot{x} = F$, where the kinematical equations are included

comb_implicit_rhs

Returns the column matrix, F , corresponding to the equations of motion in implicit form (form [2]), $M \dot{x} = F$, where the kinematical equations are included

compute_explicit_form()

If the explicit right hand side of the combined equations of motion is to provided upon initialization, this method will calculate it. This calculation can potentially take awhile to compute.

constant_symbols()

Returns a column matrix containing all of the symbols in the system that do not depend on time

coordinates

Returns the column matrix of the generalized coordinates

dyn_implicit_mat

Returns the matrix, M , corresponding to the dynamic equations in implicit form, $M \dot{x} = F$, where the kinematical equations are not included

dyn_implicit_rhs

Returns the column matrix, F , corresponding to the dynamic equations in implicit form, $M \dot{x} = F$, where the kinematical equations are not included

dynamic_symbols()

Returns a column matrix containing all of the symbols in the system that depend on time

kin_explicit_rhs

Returns the right hand side of the kinematical equations in explicit form, $q' = G$

loads

Returns the loads in the system

speeds

Returns the column matrix of generalized speeds

states

Returns the column matrix of the state variables

Linearization (Docstrings)**Linearizer**

```
class sympy.physics.mechanics.linearize.Linearizer(f_0, f_1, f_2, f_3, f_4, f_c,
                                                    f_v, f_a, q, u, q_i=None,
                                                    q_d=None, u_i=None,
                                                    u_d=None, r=None,
                                                    lams=None)
```

This object holds the general model form for a dynamic system. This model is used for computing the linearized form of the system, while properly dealing with constraints leading to dependent coordinates and speeds.

Attributes

<code>f_0, f_1, f_2, f_3, f_4, f_c, f_v, f_a</code>	(Matrix) Matrices holding the general system form.
<code>q, u, r</code>	(Matrix) Matrices holding the generalized coordinates, speeds, and input vectors.
<code>q_i, u_i</code>	(Matrix) Matrices of the independent generalized coordinates and speeds.
<code>q_d, u_d</code>	(Matrix) Matrices of the dependent generalized coordinates and speeds.
<code>perm_mat</code>	(Matrix) Permutation matrix such that $[q_{ind}, u_{ind}]^T = perm\_mat*[q, u]^T$

linearize(op_point=None, A_and_B=False, simplify=False)

Linearize the system about the operating point. Note that `q_op`, `u_op`, `qd_op`, `ud_op` must satisfy the equations of motion. These may be either symbolic or numeric.

Parameters `op_point` : dict or iterable of dicts, optional

Dictionary or iterable of dictionaries containing the operating point conditions. These will be substituted in to the linearized system before the linearization is complete. Leave blank if you want a completely symbolic form. Note that any reduction in symbols (whether substituted for numbers or expressions with a common parameter) will result in faster runtime.

A_and_B : bool, optional

If `A_and_B=False` (default), (M, A, B) is returned for forming $[M]*[q, u]^T = [A]*[q_{ind}, u_{ind}]^T + [B]r$. If `A_and_B=True`, (A, B) is returned for forming $dx = [A]x + [B]r$, where $x = [q_{ind}, u_{ind}]^T$.

simplify : bool, optional

Determines if returned values are simplified before return. For large expressions this may be time consuming. Default is False.

Potential Issues

Note that the process of solving with `A_and_B=True` is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default `A_and_B=False`, returning `M`, `A`, and `B`. More values may then be substituted in to these matrices later on. The state space form can then be found as $A = P.T*M.LUsolve(A)$, $B = P.T*M.LUsolve(B)$, where $P = Linearizer.perm\_mat$.

Expression Manipulation (Docstrings)

msubs

`sympy.physics.mechanics.msubs`(expr, *sub_dicts, **kwargs)

A custom subs for use on expressions derived in `physics.mechanics`.

Traverses the expression tree once, performing the subs found in `sub_dicts`. Terms inside Derivative expressions are ignored:


```
>>> from sympy.physics.mechanics import dynamicsymbols, msubs
>>> x = dynamicsymbols('x')
>>> msubs(x.diff() + x, {x: 1})
Derivative(x(t), t) + 1
```

Note that `sub_dicts` can be a single dictionary, or several dictionaries:

```
>>> x, y, z = dynamicsymbols('x, y, z')
>>> sub1 = {x: 1, y: 2}
>>> sub2 = {z: 3, x.diff(): 4}
>>> msubs(x.diff() + x + y + z, sub1, sub2)
10
```

If `smart=True` (default `False`), also checks for conditions that may result in `nan`, but if simplified would yield a valid expression. For example:

```
>>> from sympy import sin, tan
>>> (sin(x)/tan(x)).subs(x, 0)
nan
>>> msubs(sin(x)/tan(x), {x: 0}, smart=True)
1
```

It does this by first replacing all `tan` with `sin/cos`. Then each node is traversed. If the node is a fraction, `subs` is first evaluated on the denominator. If this results in 0, simplification of the entire fraction is attempted. Using this selective simplification, only subexpressions that result in `1/0` are targeted, resulting in faster performance.

find_dynamicsymbols

`sympy.physics.mechanics.find_dynamicsymbols(expression, exclude=None, reference_frame=None)`

Find all `dynamicsymbols` in expression.

If the optional `exclude` kwarg is used, only `dynamicsymbols` not in the iterable `exclude` are returned. If we intend to apply this function on a vector, the optional “`reference_frame`” is also used to inform about the corresponding frame with respect to which the dynamic symbols of the given vector is to be determined.

Parameters `expression` : sympy expression

exclude : iterable of `dynamicsymbols`, optional

reference_frame : `ReferenceFrame`, optional

The frame with respect to which the dynamic symbols of the given vector is to be determined.

Examples

```
>>> from sympy.physics.mechanics import dynamicsymbols, find_dynamicsymbols
>>> from sympy.physics.mechanics import ReferenceFrame
>>> x, y = dynamicsymbols('x, y')
>>> expr = x + x.diff()*y
>>> find_dynamicsymbols(expr)
{x(t), y(t), Derivative(x(t), t)}
```

(continues on next page)

(continued from previous page)

```
>>> find_dynamicsymbols(expr, exclude=[x, y])
{Derivative(x(t), t)}
>>> a, b, c = dynamicsymbols('a, b, c')
>>> A = ReferenceFrame('A')
>>> v = a * A.x + b * A.y + c * A.z
>>> find_dynamicsymbols(v, reference_frame=A)
{a(t), b(t), c(t)}
```

Printing (Docstrings)

mechanics_printing

This function is the same as `physics.vector`'s `time_derivative_printing`.

mprint

This function is the same as `physics.vector`'s `vprint`.

mpprint

This function is the same as `physics.vector`'s `vpprint`.

mlatex

This function is the same as `physics.vector`'s `vlatex`.

Body (Docstrings)

Body

class `sympy.physics.mechanics.body.Body`(name, masscenter=None, mass=None, frame=None, central_inertia=None)

Body is a common representation of either a `RigidBody` or a `Particle` SymPy object depending on what is passed in during initialization. If a mass is passed in and `central_inertia` is left as `None`, the `Particle` object is created. Otherwise a `RigidBody` object will be created.

The attributes that `Body` possesses will be the same as a `Particle` instance or a `RigidBody` instance depending on which was created. Additional attributes are listed below.

Parameters `name` : String

Defines the name of the body. It is used as the base for defining body specific properties.

masscenter : Point, optional

A point that represents the center of mass of the body or particle. If no point is given, a point is generated.

mass : Sympifyable, optional

A Sympifyable object which represents the mass of the body. If no mass is passed, one is generated.

frame : ReferenceFrame, optional

The ReferenceFrame that represents the reference frame of the body. If no frame is given, a frame is generated.

central_inertia : Dyadic, optional

Central inertia dyadic of the body. If none is passed while creating RigidBody, a default inertia is generated.

Examples

Default behaviour. This results in the creation of a RigidBody object for which the mass, mass center, frame and inertia attributes are given default values.

```
>>> from sympy.physics.mechanics import Body
>>> body = Body('name_of_body')
```

This next example demonstrates the code required to specify all of the values of the Body object. Note this will also create a RigidBody version of the Body object.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, inertia
>>> from sympy.physics.mechanics import Body
>>> mass = Symbol('mass')
>>> masscenter = Point('masscenter')
>>> frame = ReferenceFrame('frame')
>>> ixx = Symbol('ixx')
>>> body_inertia = inertia(frame, ixx, 0, 0)
>>> body = Body('name_of_body', masscenter, mass, frame, body_inertia)
```

The minimal code required to create a Particle version of the Body object involves simply passing in a name and a mass.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> mass = Symbol('mass')
>>> body = Body('name_of_body', mass=mass)
```

The Particle version of the Body object can also receive a masscenter point and a reference frame, just not an inertia.

Attributes

name	(string) The body's name
mass-center	(Point) The point which represents the center of mass of the rigid body
frame	(ReferenceFrame) The reference frame which the body is fixed in
mass	(Sympifyable) The body's mass
inertia	((Dyadic, Point)) The body's inertia around its center of mass. This attribute is specific to the rigid body form of Body and is left undefined for the Particle form
loads	(iterable) This list contains information on the different loads acting on the Body. Forces are listed as a (point, vector) tuple and torques are listed as (reference frame, vector) tuples.

apply_force(vec, point=None)

Adds a force to a point (center of mass by default) on the body.

Parameters **vec: Vector**

Defines the force vector. Can be any vector w.r.t any frame or combinations of frames.

point: Point, optional

Defines the point on which the force is applied. Default is the Body's center of mass.

Example

The first example applies a gravitational force in the x direction of Body's frame to the body's center of mass.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> body = Body('body')
>>> g = Symbol('g')
>>> body.apply_force(body.mass * g * body.frame.x)
```

To apply force to any other point than center of mass, pass that point as well. This example applies a gravitational force to a point a distance l from the body's center of mass in the y direction. The force is again applied in the x direction.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> body = Body('body')
>>> g = Symbol('g')
>>> l = Symbol('l')
>>> point = body.masscenter.locatenew('force_point', l *
...                                   body.frame.y)
>>> body.apply_force(body.mass * g * body.frame.x, point)
```

apply_torque(vec)

Adds a torque to the body.

Parameters **vec: Vector**

Defines the torque vector. Can be any vector w.r.t any frame or combinations of frame.

Example

This example adds a simple torque around the body's z axis.

```
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> body = Body('body')
>>> T = Symbol('T')
>>> body.apply_torque(T * body.frame.z)
```

Quantum Mechanics

Abstract

Contains Docstrings of Physics-Quantum module

Quantum Functions

Anticommutator

The anti-commutator: $\{A, B\} = A*B + B*A$.

class sympy.physics.quantum.anticommutator.**AntiCommutator**

The standard anticommutator, in an unevaluated state.

Evaluating an anticommutator is defined [R430] (page 1873) as: $\{A, B\} = A*B + B*A$. This class returns the anticommutator in an unevaluated form. To evaluate the anticommutator, use the `.doit()` method.

Canonical ordering of an anticommutator is $\{A, B\}$ for $A < B$. The arguments of the anticommutator are put into canonical order using `__cmp__`. If $B < A$, then $\{A, B\}$ is returned as $\{B, A\}$.

Parameters **A** : Expr

The first argument of the anticommutator $\{A, B\}$.

B : Expr

The second argument of the anticommutator $\{A, B\}$.

References

[R430] (page 1873)

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.quantum import AntiCommutator
>>> from sympy.physics.quantum import Operator, Dagger
>>> x, y = symbols('x,y')
>>> A = Operator('A')
>>> B = Operator('B')
```

Create an anticommutator and use `doit()` to multiply them out.

```
>>> ac = AntiCommutator(A,B); ac
{A,B}
>>> ac.doit()
A*B + B*A
```

The commutator orders its arguments in canonical order:

```
>>> ac = AntiCommutator(B,A); ac
{A,B}
```

Commutative constants are factored out:

```
>>> AntiCommutator(3*x*A,x*y*B)
3*x**2*y*{A,B}
```

Adjoint operations applied to the anticommutator are properly applied to the arguments:

```
>>> Dagger(AntiCommutator(A,B))
{Dagger(A),Dagger(B)}
```

```
doit(**hints)
    Evaluate anticommutator
```

Clebsch-Gordan Coefficients

Clebsch-Gordon Coefficients.

class `sympy.physics.quantum.cg.CG`
Class for Clebsch-Gordan coefficient

Clebsch-Gordan coefficients describe the angular momentum coupling between two systems. The coefficients give the expansion of a coupled total angular momentum state and an uncoupled tensor product state. The Clebsch-Gordan coefficients are defined as [R431] (page 1873):

$$C_{j_2, m_2, j_3, m_3}^{j_1, m_1} = \langle j_1, m_1; j_2, m_2 | j_3, m_3 \rangle$$

Parameters `j1, m1, j2, m2, j3, m3` : Number, Symbol

Terms determining the angular momentum of coupled angular momentum systems.

See also:

Wigner3j (page 1649) Wigner-3j symbols

References

[R431] (page 1873)

Examples

Define a Clebsch-Gordan coefficient and evaluate its value

```
>>> from sympy.physics.quantum.cg import CG
>>> from sympy import S
>>> cg = CG(S(3)/2, S(3)/2, S(1)/2, -S(1)/2, 1, 1)
>>> cg
CG(3/2, 3/2, 1/2, -1/2, 1, 1)
>>> cg.doit()
sqrt(3)/2
```

class sympy.physics.quantum.cg.Wigner3j

Class for the Wigner-3j symbols

Wigner 3j-symbols are coefficients determined by the coupling of two angular momenta. When created, they are expressed as symbolic quantities that, for numerical parameters, can be evaluated using the `.doit()` method [R432] (page 1873).

Parameters `j1, m1, j2, m2, j3, m3` : Number, Symbol

Terms determining the angular momentum of coupled angular momentum systems.

See also:

CG (page 1648) Clebsch-Gordan coefficients

References

[R432] (page 1873)

Examples

Declare a Wigner-3j coefficient and calculate its value

```
>>> from sympy.physics.quantum.cg import Wigner3j
>>> w3j = Wigner3j(6,0,4,0,2,0)
>>> w3j
Wigner3j(6, 0, 4, 0, 2, 0)
>>> w3j.doit()
sqrt(715)/143
```

class sympy.physics.quantum.cg.Wigner6j

Class for the Wigner-6j symbols

See also:

Wigner3j (page 1649) Wigner-3j symbols

class `sympy.physics.quantum.cg.Wigner9j`

Class for the Wigner-9j symbols

See also:

[Wigner3j](#) (page 1649) Wigner-3j symbols

`sympy.physics.quantum.cg.cg_simp(e)`

Simplify and combine CG coefficients

This function uses various symmetry and properties of sums and products of Clebsch-Gordan coefficients to simplify statements involving these terms [\[R433\]](#) (page 1873).

See also:

[CG](#) (page 1648) Clebsch-Gordan coefficients

References

[\[R433\]](#) (page 1873)

Examples

Simplify the sum over $CG(a, \alpha, 0, 0, a, \alpha)$ for all α to $2*a+1$

```
>>> from sympy.physics.quantum.cg import CG, cg_simp
>>> a = CG(1,1,0,0,1,1)
>>> b = CG(1,0,0,0,1,0)
>>> c = CG(1,-1,0,0,1,-1)
>>> cg_simp(a+b+c)
3
```

Commutator

The commutator: $[A, B] = A*B - B*A$.

class `sympy.physics.quantum.commutator.Commutator`

The standard commutator, in an unevaluated state.

Evaluating a commutator is defined [\[R434\]](#) (page 1873) as: $[A, B] = A*B - B*A$. This class returns the commutator in an unevaluated form. To evaluate the commutator, use the `.doit()` method.

Canonical ordering of a commutator is $[A, B]$ for $A < B$. The arguments of the commutator are put into canonical order using `__cmp__`. If $B < A$, then $[B, A]$ is returned as $-[A, B]$.

Parameters **A** : Expr

The first argument of the commutator $[A, B]$.

B : Expr

The second argument of the commutator $[A, B]$.

References

[R434] (page 1873)

Examples

```
>>> from sympy.physics.quantum import Commutator, Dagger, Operator
>>> from sympy.abc import x, y
>>> A = Operator('A')
>>> B = Operator('B')
>>> C = Operator('C')
```

Create a commutator and use `.doit()` to evaluate it:

```
>>> comm = Commutator(A, B)
>>> comm
[A,B]
>>> comm.doit()
A*B - B*A
```

The commutator orders its arguments in canonical order:

```
>>> comm = Commutator(B, A); comm
-[A,B]
```

Commutative constants are factored out:

```
>>> Commutator(3*x*A, x*y*B)
3*x**2*y*[A,B]
```

Using `.expand(commutator=True)`, the standard commutator expansion rules can be applied:

```
>>> Commutator(A+B, C).expand(commutator=True)
[A,C] + [B,C]
>>> Commutator(A, B+C).expand(commutator=True)
[A,B] + [A,C]
>>> Commutator(A*B, C).expand(commutator=True)
[A,C]*B + A*[B,C]
>>> Commutator(A, B*C).expand(commutator=True)
[A,B]*C + B*[A,C]
```

Adjoint operations applied to the commutator are properly applied to the arguments:

```
>>> Dagger(Commutator(A, B))
-[Dagger(A), Dagger(B)]
```

doit(hints)**
Evaluate commutator

Constants

Constants (like `hbar`) related to quantum mechanics.

Dagger

Hermitian conjugation.

class sympy.physics.quantum.dagger.Dagger

General Hermitian conjugate operation.

Take the Hermetian conjugate of an argument [R435] (page 1873). For matrices this operation is equivalent to transpose and complex conjugate [R436] (page 1873).

Parameters **arg** : Expr

The sympy expression that we want to take the dagger of.

References

[R435] (page 1873), [R436] (page 1873)

Examples

Dagging various quantum objects:

```
>>> from sympy.physics.quantum.dagger import Dagger
>>> from sympy.physics.quantum.state import Ket, Bra
>>> from sympy.physics.quantum.operator import Operator
>>> Dagger(Ket('psi'))
<psi|
>>> Dagger(Bra('phi'))
|phi>
>>> Dagger(Operator('A'))
Dagger(A)
```

Inner and outer products:

```
>>> from sympy.physics.quantum import InnerProduct, OuterProduct
>>> Dagger(InnerProduct(Bra('a'), Ket('b')))
<b|a>
>>> Dagger(OuterProduct(Ket('a'), Bra('b')))
|b><a|
```

Powers, sums and products:

```
>>> A = Operator('A')
>>> B = Operator('B')
>>> Dagger(A*B)
Dagger(B)*Dagger(A)
>>> Dagger(A+B)
Dagger(A) + Dagger(B)
>>> Dagger(A**2)
Dagger(A)**2
```

Dagger also seamlessly handles complex numbers and matrices:

```
>>> from sympy import Matrix, I
>>> m = Matrix([[1,I],[2,I]])
>>> m
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, I],
[2, I]])
>>> Dagger(m)
Matrix([
[ 1,  2],
[-I, -I]])
```

Inner Product

Symbolic inner product.

class `sympy.physics.quantum.innerproduct.InnerProduct`
 An unevaluated inner product between a Bra and a Ket [1].

Parameters **bra** : BraBase or subclass

The bra on the left side of the inner product.

ket : KetBase or subclass

The ket on the right side of the inner product.

References

[R439] (page 1873)

Examples

Create an InnerProduct and check its properties:

```
>>> from sympy.physics.quantum import Bra, Ket, InnerProduct
>>> b = Bra('b')
>>> k = Ket('k')
>>> ip = b*k
>>> ip
<b|k>
>>> ip.bra
<b|
>>> ip.ket
|k>
```

In simple products of kets and bras inner products will be automatically identified and created:

```
>>> b*k
<b|k>
```

But in more complex expressions, there is ambiguity in whether inner or outer products should be created:

```
>>> k*b*k*b
|k><b|*|k><b|
```

A user can force the creation of an inner product in a complex expression by using parentheses to group the bra and ket:

```
>>> k*(b*k)*b
<b|k>|k>*<b|
```

Notice how the inner product $\langle b|k\rangle$ moved to the left of the expression because inner products are commutative complex numbers.

Tensor Product

Abstract tensor product.

class sympy.physics.quantum.tensorproduct.TensorProduct

The tensor product of two or more arguments.

For matrices, this uses `matrix_tensor_product` to compute the Kronecker or tensor product matrix. For other objects a symbolic `TensorProduct` instance is returned. The tensor product is a non-commutative multiplication that is used primarily with operators and states in quantum mechanics.

Currently, the tensor product distinguishes between commutative and non-commutative arguments. Commutative arguments are assumed to be scalars and are pulled out in front of the `TensorProduct`. Non-commutative arguments remain in the resulting `TensorProduct`.

Parameters `args` : tuple

A sequence of the objects to take the tensor product of.

Examples

Start with a simple tensor product of sympy matrices:

```
>>> from sympy import I, Matrix, symbols
>>> from sympy.physics.quantum import TensorProduct

>>> m1 = Matrix([[1,2],[3,4]])
>>> m2 = Matrix([[1,0],[0,1]])
>>> TensorProduct(m1, m2)
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2],
[3, 0, 4, 0],
[0, 3, 0, 4]])
>>> TensorProduct(m2, m1)
Matrix([
[1, 2, 0, 0],
[3, 4, 0, 0],
[0, 0, 1, 2],
[0, 0, 3, 4]])
```

We can also construct tensor products of non-commutative symbols:

```
>>> from sympy import Symbol
>>> A = Symbol('A', commutative=False)
```

(continues on next page)

(continued from previous page)

```
>>> B = Symbol('B', commutative=False)
>>> tp = TensorProduct(A, B)
>>> tp
AxB
```

We can take the dagger of a tensor product (note the order does NOT reverse like the dagger of a normal product):

```
>>> from sympy.physics.quantum import Dagger
>>> Dagger(tp)
Dagger(A)xDagger(B)
```

Expand can be used to distribute a tensor product across addition:

```
>>> C = Symbol('C', commutative=False)
>>> tp = TensorProduct(A+B, C)
>>> tp
(A + B)xC
>>> tp.expand(tensorproduct=True)
AxC + BxC
```

`sympy.physics.quantum.tensorproduct.tensor_product_simp(e, **hints)`

Try to simplify and combine TensorProducts.

In general this will try to pull expressions inside of TensorProducts. It currently only works for relatively simple cases where the products have only scalars, raw TensorProducts, not Add, Pow, Commutators of TensorProducts. It is best to see what it does by showing examples.

Examples

```
>>> from sympy.physics.quantum import tensor_product_simp
>>> from sympy.physics.quantum import TensorProduct
>>> from sympy import Symbol
>>> A = Symbol('A', commutative=False)
>>> B = Symbol('B', commutative=False)
>>> C = Symbol('C', commutative=False)
>>> D = Symbol('D', commutative=False)
```

First see what happens to products of tensor products:

```
>>> e = TensorProduct(A,B)*TensorProduct(C,D)
>>> e
AxBxCxD
>>> tensor_product_simp(e)
(A*C)x(B*D)
```

This is the core logic of this function, and it works inside, powers, sums, commutators and anticommutators as well:

```
>>> tensor_product_simp(e**2)
(A*C)x(B*D)**2
```

States and Operators

Cartesian Operators and States

Operators and states for 1D cartesian position and momentum.

TODO:

- Add 3D classes to mappings in operatorset.py

class sympy.physics.quantum.cartesian.X0p

1D cartesian position operator.

class sympy.physics.quantum.cartesian.Y0p

Y cartesian coordinate operator (for 2D or 3D systems)

class sympy.physics.quantum.cartesian.Z0p

Z cartesian coordinate operator (for 3D systems)

class sympy.physics.quantum.cartesian.Px0p

1D cartesian momentum operator.

class sympy.physics.quantum.cartesian.XKet

1D cartesian position eigenket.

position

The position of the state.

class sympy.physics.quantum.cartesian.XBra

1D cartesian position eigenbra.

position

The position of the state.

class sympy.physics.quantum.cartesian.PxKet

1D cartesian momentum eigenket.

momentum

The momentum of the state.

class sympy.physics.quantum.cartesian.PxBra

1D cartesian momentum eigenbra.

momentum

The momentum of the state.

class sympy.physics.quantum.cartesian.PositionState3D

Base class for 3D cartesian position eigenstates

position_x

The x coordinate of the state

position_y

The y coordinate of the state

position_z

The z coordinate of the state

class sympy.physics.quantum.cartesian.PositionKet3D

3D cartesian position eigenket

class sympy.physics.quantum.cartesian.PositionBra3D

3D cartesian position eigenbra

Hilbert Space

Hilbert spaces for quantum mechanics.

Authors: * Brian Granger * Matt Curry

class `sympy.physics.quantum.hilbert.HilbertSpace`

An abstract Hilbert space for quantum mechanics.

In short, a Hilbert space is an abstract vector space that is complete with inner products defined [R437] (page 1873).

References

[R437] (page 1873)

Examples

```
>>> from sympy.physics.quantum.hilbert import HilbertSpace
>>> hs = HilbertSpace()
>>> hs
H
```

dimension

Return the Hilbert dimension of the space.

class `sympy.physics.quantum.hilbert.ComplexSpace`

Finite dimensional Hilbert space of complex vectors.

The elements of this Hilbert space are n-dimensional complex valued vectors with the usual inner product that takes the complex conjugate of the vector on the right.

A classic example of this type of Hilbert space is spin-1/2, which is `ComplexSpace(2)`. Generalizing to spin-s, the space is `ComplexSpace(2*s+1)`. Quantum computing with N qubits is done with the direct product space `ComplexSpace(2)**N`.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.quantum.hilbert import ComplexSpace
>>> c1 = ComplexSpace(2)
>>> c1
C(2)
>>> c1.dimension
2
```

```
>>> n = symbols('n')
>>> c2 = ComplexSpace(n)
>>> c2
C(n)
>>> c2.dimension
n
```

class sympy.physics.quantum.hilbert.L2

The Hilbert space of square integrable functions on an interval.

An L2 object takes in a single sympy Interval argument which represents the interval its functions (vectors) are defined on.

Examples

```
>>> from sympy import Interval, oo
>>> from sympy.physics.quantum.hilbert import L2
>>> hs = L2(Interval(0,oo))
>>> hs
L2(Interval(0, oo))
>>> hs.dimension
oo
>>> hs.interval
Interval(0, oo)
```

class sympy.physics.quantum.hilbert.FockSpace

The Hilbert space for second quantization.

Technically, this Hilbert space is a infinite direct sum of direct products of single particle Hilbert spaces [R438] (page 1873). This is a mess, so we have a class to represent it directly.

References

[R438] (page 1873)

Examples

```
>>> from sympy.physics.quantum.hilbert import FockSpace
>>> hs = FockSpace()
>>> hs
F
>>> hs.dimension
oo
```

Operator

Quantum mechanical operators.

TODO:

- Fix early 0 in apply_operators.
- Debug and test apply_operators.
- Get cse working with classes in this file.
- Doctests and documentation of special methods for InnerProduct, Commutator, Anti-Commutator, represent, apply_operators.

class sympy.physics.quantum.operator.Operator

Base class for non-commuting quantum operators.

An operator maps between quantum states [R440] (page 1873). In quantum mechanics, observables (including, but not limited to, measured physical values) are represented as Hermitian operators [R441] (page 1873).

Parameters args : tuple

The list of numbers or parameters that uniquely specify the operator.

For time-dependent operators, this will include the time.

References

[R440] (page 1873), [R441] (page 1873)

Examples

Create an operator and examine its attributes:

```
>>> from sympy.physics.quantum import Operator
>>> from sympy import symbols, I
>>> A = Operator('A')
>>> A
A
>>> A.hilbert_space
H
>>> A.label
(A,)
>>> A.is_commutative
False
```

Create another operator and do some arithmetic operations:

```
>>> B = Operator('B')
>>> C = 2*A*A + I*B
>>> C
2*A**2 + I*B
```

Operators don't commute:

```
>>> A.is_commutative
False
>>> B.is_commutative
False
>>> A*B == B*A
False
```

Polynomials of operators respect the commutation properties:

```
>>> e = (A+B)**3
>>> e.expand()
A*B*A + A*B**2 + A**2*B + A**3 + B*A*B + B*A**2 + B**2*A + B**3
```

Operator inverses are handle symbolically:

```
>>> A.inv()
A**(-1)
>>> A*A.inv()
1
```

class sympy.physics.quantum.operator.HermitianOperator

A Hermitian operator that satisfies $H == \text{Dagger}(H)$.

Parameters *args* : tuple

The list of numbers or parameters that uniquely specify the operator.
For time-dependent operators, this will include the time.

Examples

```
>>> from sympy.physics.quantum import Dagger, HermitianOperator
>>> H = HermitianOperator('H')
>>> Dagger(H)
H
```

class sympy.physics.quantum.operator.UnitaryOperator

A unitary operator that satisfies $U*\text{Dagger}(U) == 1$.

Parameters *args* : tuple

The list of numbers or parameters that uniquely specify the operator.
For time-dependent operators, this will include the time.

Examples

```
>>> from sympy.physics.quantum import Dagger, UnitaryOperator
>>> U = UnitaryOperator('U')
>>> U*Dagger(U)
1
```

class sympy.physics.quantum.operator.IdentityOperator(*args, **hints)

An identity operator I that satisfies $op * I == I * op == op$ for any operator op .

Parameters *N* : Integer

Optional parameter that specifies the dimension of the Hilbert space of operator. This is used when generating a matrix representation.

Examples

```
>>> from sympy.physics.quantum import IdentityOperator
>>> IdentityOperator()
I
```

class sympy.physics.quantum.operator.OuterProduct

An unevaluated outer product between a ket and bra.

This constructs an outer product between any subclass of `KetBase` and `BraBase` as $|a\rangle\langle b|$. An `OuterProduct` inherits from `Operator` as they act as operators in quantum expressions. For reference see [\[R442\]](#) (page 1873).

Parameters **ket** : KetBase

The ket on the left side of the outer product.

bar : BraBase

The bra on the right side of the outer product.

References

[R442] (page 1873)

Examples

Create a simple outer product by hand and take its dagger:

```
>>> from sympy.physics.quantum import Ket, Bra, OuterProduct, Dagger
>>> from sympy.physics.quantum import Operator

>>> k = Ket('k')
>>> b = Bra('b')
>>> op = OuterProduct(k, b)
>>> op
|k><b|
>>> op.hilbert_space
H
>>> op.ket
|k>
>>> op.bra
<b|
>>> Dagger(op)
|b><k|
```

In simple products of kets and bras outer products will be automatically identified and created:

```
>>> k*b
|k><b|
```

But in more complex expressions, outer products are not automatically created:

```
>>> A = Operator('A')
>>> A*k*b
A*|k>*<b|
```

A user can force the creation of an outer product in a complex expression by using parentheses to group the ket and bra:

```
>>> A*(k*b)
A*|k><b|
```

bra

Return the bra on the right side of the outer product.

ket

Return the ket on the left side of the outer product.

class sympy.physics.quantum.operator.DifferentialOperator

An operator for representing the differential operator, i.e. d/dx

It is initialized by passing two arguments. The first is an arbitrary expression that involves a function, such as `Derivative(f(x), x)`. The second is the function (e.g. `f(x)`) which we are to replace with the Wavefunction that this `DifferentialOperator` is applied to.

Parameters `expr` : Expr

The arbitrary expression which the appropriate Wavefunction is to be substituted into

func : Expr

A function (e.g. `f(x)`) which is to be replaced with the appropriate Wavefunction when this `DifferentialOperator` is applied

Examples

You can define a completely arbitrary expression and specify where the Wavefunction is to be substituted

```
>>> from sympy import Derivative, Function, Symbol
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy.physics.quantum.qapply import qapply
>>> f = Function('f')
>>> x = Symbol('x')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> w = Wavefunction(x**2, x)
>>> d.function
f(x)
>>> d.variables
(x,)
>>> qapply(d*w)
Wavefunction(2, x)
```

expr

Returns the arbitrary expression which is to have the Wavefunction substituted into it

Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.expr
Derivative(f(x), x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
...                          Derivative(f(x, y), y), f(x, y))
>>> d.expr
Derivative(f(x, y), x) + Derivative(f(x, y), y)
```

free_symbols

Return the free symbols of the expression.

function

Returns the function which is to be replaced with the Wavefunction

Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.function
f(x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
...                           Derivative(f(x, y), y), f(x, y))
>>> d.function
f(x, y)
```

variables

Returns the variables with which the function in the specified arbitrary expression is evaluated

Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Symbol, Function, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> d.variables
(x,)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
...                           Derivative(f(x, y), y), f(x, y))
>>> d.variables
(x, y)
```

Operator/State Helper Functions

A module for mapping operators to their corresponding eigenstates and vice versa

It contains a global dictionary with eigenstate-operator pairings. If a new state-operator pair is created, this dictionary should be updated as well.

It also contains functions `operators_to_state` and `state_to_operators` for mapping between the two. These can handle both classes and instances of operators and states. See the individual function descriptions for details.

TODO List: - Update the dictionary with a complete list of state-operator pairs

`sympy.physics.quantum.operatorset.operators_to_state(operators, **options)`

Returns the eigenstate of the given operator or set of operators

A global function for mapping operator classes to their associated states. It takes either an `Operator` or a set of operators and returns the state associated with these.

This function can handle both instances of a given operator or just the class itself (i.e. both `XOp()` and `XOp`)

There are multiple use cases to consider:

1) A class or set of classes is passed: First, we try to instantiate default instances for these operators. If this fails, then the class is simply returned. If we succeed in instantiating default instances, then we try to call `state._operators_to_state` on the operator instances. If this fails, the class is returned. Otherwise, the instance returned by `_operators_to_state` is returned.

2) An instance or set of instances is passed: In this case, `state._operators_to_state` is called on the instances passed. If this fails, a state class is returned. If the method returns an instance, that instance is returned.

In both cases, if the operator class or set does not exist in the `state_mapping` dictionary, `None` is returned.

Parameters arg: Operator or set

The class or instance of the operator or set of operators to be mapped to a state

Examples

```
>>> from sympy.physics.quantum.cartesian import XOp, PxOp
>>> from sympy.physics.quantum.operatorset import operators_to_state
>>> from sympy.physics.quantum.operator import Operator
>>> operators_to_state(XOp)
|x>
>>> operators_to_state(XOp())
|x>
>>> operators_to_state(PxOp)
|px>
>>> operators_to_state(PxOp())
|px>
>>> operators_to_state(Operator)
|psi>
>>> operators_to_state(Operator())
|psi>
```

`sympy.physics.quantum.operatorset.state_to_operators(state, **options)`

Returns the operator or set of operators corresponding to the given eigenstate

A global function for mapping state classes to their associated operators or sets of operators. It takes either a state class or instance.

This function can handle both instances of a given state or just the class itself (i.e. both `XKet()` and `XKet`)

There are multiple use cases to consider:

1) A state class is passed: In this case, we first try instantiating a default instance of the class. If this succeeds, then we try to call `state._state_to_operators` on that instance. If the creation of the default instance or if the calling of `_state_to_operators` fails, then

either an operator class or set of operator classes is returned. Otherwise, the appropriate operator instances are returned.

2) A state instance is returned: Here, `state._state_to_operators` is called for the instance. If this fails, then a class or set of operator classes is returned. Otherwise, the instances are returned.

In either case, if the state's class does not exist in `state_mapping`, `None` is returned.

Parameters `arg`: **StateBase class or instance (or subclasses)**

The class or instance of the state to be mapped to an operator or set of operators

Examples

```
>>> from sympy.physics.quantum.cartesian import XKet, PxKet, XBra, PxBra
>>> from sympy.physics.quantum.operatorset import state_to_operators
>>> from sympy.physics.quantum.state import Ket, Bra
>>> state_to_operators(XKet)
X
>>> state_to_operators(XKet())
X
>>> state_to_operators(PxKet)
Px
>>> state_to_operators(PxKet())
Px
>>> state_to_operators(PxBra)
Px
>>> state_to_operators(XBra)
X
>>> state_to_operators(Ket)
0
>>> state_to_operators(Bra)
0
```

Qapply

Logic for applying operators to states.

Todo: * Sometimes the final result needs to be expanded, we should do this by hand.

`sympy.physics.quantum.qapply.qapply(e, **options)`

Apply operators to states in a quantum expression.

Parameters `e` : Expr

The expression containing operators and states. This expression tree will be walked to find operators acting on states symbolically.

options : dict

A dict of key/value pairs that determine how the operator actions are carried out.

The following options are valid:

- `dagger`: try to apply Dagger operators to the left (default: `False`).

- `ip_doit`: call `.doit()` in inner products when they are encountered (default: `True`).

Returns `e` : Expr

The original expression, but with the operators applied to states.

Examples

```
>>> from sympy.physics.quantum import qapply, Ket, Bra
>>> b = Bra('b')
>>> k = Ket('k')
>>> A = k * b
>>> A
|k><b|
>>> qapply(A * b.dual / (b * b.dual))
|k>
>>> qapply(k.dual * A / (k.dual * k), dagger=True)
<b|
>>> qapply(k.dual * A / (k.dual * k))
<k|*|k><b|/<k|k>
```

Represent

Logic for representing operators in state in various bases.

TODO:

- Get represent working with continuous hilbert spaces.
- Document default basis functionality.

`sympy.physics.quantum.represent.represent(expr, **options)`

Represent the quantum expression in the given basis.

In quantum mechanics abstract states and operators can be represented in various basis sets. Under this operation the follow transforms happen:

- Ket -> column vector or function
- Bra -> row vector of function
- Operator -> matrix or differential operator

This function is the top-level interface for this action.

This function walks the sympy expression tree looking for `QExpr` instances that have a `_represent` method. This method is then called and the object is replaced by the representation returned by this method. By default, the `_represent` method will dispatch to other methods that handle the representation logic for a particular basis set. The naming convention for these methods is the following:

```
def _represent_FooBasis(self, e, basis, **options)
```

This function will have the logic for representing instances of its class in the basis set having a class named `FooBasis`.

Parameters `expr` : Expr

The expression to represent.

basis : Operator, basis set

An object that contains the information about the basis set. If an operator is used, the basis is assumed to be the orthonormal eigenvectors of that operator. In general though, the basis argument can be any object that contains the basis set information.

options : dict

Key/value pairs of options that are passed to the underlying method that finds the representation. These options can be used to control how the representation is done. For example, this is where the size of the basis set would be set.

Returns **e** : Expr

The SymPy expression of the represented quantum expression.

Examples

Here we subclass `Operator` and `Ket` to create the z-spin operator and its spin 1/2 up eigenstate. By defining the `_represent_Sz0p` method, the ket can be represented in the z-spin basis.

```
>>> from sympy.physics.quantum import Operator, represent, Ket
>>> from sympy import Matrix
```

```
>>> class SzUpKet(Ket):
...     def _represent_Sz0p(self, basis, **options):
...         return Matrix([1,0])
...
>>> class Sz0p(Operator):
...     pass
...
>>> sz = Sz0p('Sz')
>>> up = SzUpKet('up')
>>> represent(up, basis=sz)
Matrix([
[1],
[0]])
```

Here we see an example of representations in a continuous basis. We see that the result of representing various combinations of cartesian position operators and kets give us continuous expressions involving DiracDelta functions.

```
>>> from sympy.physics.quantum.cartesian import X0p, XKet, XBra
>>> X = X0p()
>>> x = XKet()
>>> y = XBra('y')
>>> represent(X*x)
x*DiracDelta(x - x_2)
>>> represent(X*x*y)
x*DiracDelta(x - x_3)*DiracDelta(x_1 - y)
```

`sympy.physics.quantum.represent.rep_innerproduct(expr, **options)`

Returns an innerproduct like representation (e.g. $\langle x' | x \rangle$) for the given state.

Attempts to calculate inner product with a bra from the specified basis. Should only be passed an instance of `KetBase` or `BraBase`

Parameters **expr** : KetBase or BraBase

The expression to be represented

Examples

```
>>> from sympy.physics.quantum.represent import rep_innerproduct
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> rep_innerproduct(XKet())
DiracDelta(x - x_1)
>>> rep_innerproduct(XKet(), basis=PxOp())
sqrt(2)*exp(-I*px_1*x/hbar)/(2*sqrt(hbar)*sqrt(pi))
>>> rep_innerproduct(PxKet(), basis=XOp())
sqrt(2)*exp(I*px*x_1/hbar)/(2*sqrt(hbar)*sqrt(pi))
```

`sympy.physics.quantum.represent.rep_expectation(expr, **options)`

Returns an $\langle x' | A | x \rangle$ type representation for the given operator.

Parameters **expr** : Operator

Operator to be represented in the specified basis

Examples

```
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> from sympy.physics.quantum.represent import rep_expectation
>>> rep_expectation(XOp())
x_1*DiracDelta(x_1 - x_2)
>>> rep_expectation(XOp(), basis=PxOp())
<px_2|*X*|px_1>
>>> rep_expectation(XOp(), basis=PxKet())
<px_2|*X*|px_1>
```

`sympy.physics.quantum.represent.integrate_result(orig_expr, result, **options)`

Returns the result of integrating over any unities ($|x\rangle\langle x|$) in the given expression. Intended for integrating over the result of representations in continuous bases.

This function integrates over any unities that may have been inserted into the quantum expression and returns the result. It uses the interval of the Hilbert space of the basis state passed to it in order to figure out the limits of integration. The unities option must be specified for this to work.

Note: This is mostly used internally by `represent()`. Examples are given merely to show the use cases.

Parameters **orig_expr** : quantum expression

The original expression which was to be represented

result: Expr

The resulting representation that we wish to integrate over

Examples

```
>>> from sympy import symbols, DiracDelta
>>> from sympy.physics.quantum.represent import integrate_result
>>> from sympy.physics.quantum.cartesian import XOp, XKet
>>> x_ket = XKet()
>>> X_op = XOp()
>>> x, x_1, x_2 = symbols('x, x_1, x_2')
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2))
x*DiracDelta(x - x_1)*DiracDelta(x_1 - x_2)
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2),
...                  unities=[1])
x*DiracDelta(x - x_2)
```

`sympy.physics.quantum.represent.get_basis(expr, **options)`

Returns a basis state instance corresponding to the basis specified in `options=s`. If no basis is specified, the function tries to form a default basis state of the given expression.

There are three behaviors:

1. The basis specified in `options` is already an instance of `StateBase`. If this is the case, it is simply returned. If the class is specified but not an instance, a default instance is returned.
2. The basis specified is an operator or set of operators. If this is the case, the `operator_to_state` mapping method is used.
3. No basis is specified. If `expr` is a state, then a default instance of its class is returned. If `expr` is an operator, then it is mapped to the corresponding state. If it is neither, then we cannot obtain the basis state.

If the basis cannot be mapped, then it is not changed.

This will be called from within `represent`, and `represent` will only pass `QExpr`'s.

TODO (?): Support for Muls and other types of expressions?

Parameters `expr` : Operator or `StateBase`

Expression whose basis is sought

Examples

```
>>> from sympy.physics.quantum.represent import get_basis
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> x = XKet()
>>> X = XOp()
>>> get_basis(x)
|x>
>>> get_basis(X)
|x>
>>> get_basis(x, basis=PxOp())
|px>
>>> get_basis(x, basis=PxKet)
|px>
```

`sympy.physics.quantum.represent.enumerate_states(*args, **options)`

Returns instances of the given state with dummy indices appended

Operates in two different modes:

1. Two arguments are passed to it. The first is the base state which is to be indexed, and the second argument is a list of indices to append.
2. Three arguments are passed. The first is again the base state to be indexed. The second is the start index for counting. The final argument is the number of kets you wish to receive.

Tries to call `state._enumerate_state`. If this fails, returns an empty list

Parameters `args` : list

See list of operation modes above for explanation

Examples

```
>>> from sympy.physics.quantum.cartesian import XBra, XKet
>>> from sympy.physics.quantum.represent import enumerate_states
>>> test = XKet('foo')
>>> enumerate_states(test, 1, 3)
[|foo_1>, |foo_2>, |foo_3>]
>>> test2 = XBra('bar')
>>> enumerate_states(test2, [4, 5, 10])
[<bar_4|, <bar_5|, <bar_10|]
```

Spin

Quantum mechanical angular momentum.

class `sympy.physics.quantum.spin.Rotation`

Wigner D operator in terms of Euler angles.

Defines the rotation operator in terms of the Euler angles defined by the z-y-z convention for a passive transformation. That is the coordinate axes are rotated first about the z-axis, giving the new x'-y'-z' axes. Then this new coordinate system is rotated about the new y'-axis, giving new x''-y''-z'' axes. Then this new coordinate system is rotated about the z''-axis. Conventions follow those laid out in [\[R443\]](#) (page 1873).

Parameters `alpha` : Number, Symbol

First Euler Angle

beta : Number, Symbol

Second Euler angle

gamma : Number, Symbol

Third Euler angle

See also:

WignerD (page 1672) Symbolic Wigner-D function

D (page 1671) Wigner-D function

d (page 1671) Wigner small-d function

References

[\[R443\]](#) (page 1873)

Examples

A simple example rotation operator:

```
>>> from sympy import pi
>>> from sympy.physics.quantum.spin import Rotation
>>> Rotation(pi, 0, pi/2)
R(pi,0,pi/2)
```

With symbolic Euler angles and calculating the inverse rotation operator:

```
>>> from sympy import symbols
>>> a, b, c = symbols('a b c')
>>> Rotation(a, b, c)
R(a,b,c)
>>> Rotation(a, b, c).inverse()
R(-c,-b,-a)
```

classmethod **D**(j, m, mp, alpha, beta, gamma)

Wigner D-function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters.

Parameters **j** : Number

Total angular momentum

m : Number

Eigenvalue of angular momentum along axis after rotation

mp : Number

Eigenvalue of angular momentum along rotated axis

alpha : Number, Symbol

First Euler angle of rotation

beta : Number, Symbol

Second Euler angle of rotation

gamma : Number, Symbol

Third Euler angle of rotation

See also:

WignerD ([page 1672](#)) Symbolic Wigner-D function

Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> alpha, beta, gamma = symbols('alpha beta gamma')
>>> Rotation.D(1, 1, 0, pi, pi/2, -pi)
WignerD(1, 1, 0, pi, pi/2, -pi)
```

classmethod `d(j, m, mp, beta)`

Wigner small-d function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters with the alpha and gamma angles given as 0.

Parameters `j` : Number

Total angular momentum

`m` : Number

Eigenvalue of angular momentum along axis after rotation

`mp` : Number

Eigenvalue of angular momentum along rotated axis

`beta` : Number, Symbol

Second Euler angle of rotation

See also:

[WignerD \(page 1672\)](#) Symbolic Wigner-D function

Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> beta = symbols('beta')
>>> Rotation.d(1, 1, 0, pi/2)
WignerD(1, 1, 0, 0, pi/2, 0)
```

class `sympy.physics.quantum.spin.WignerD`

Wigner-D function

The Wigner D-function gives the matrix elements of the rotation operator in the jm-representation. For the Euler angles α , β , γ , the D-function is defined such that:

$$\langle j, m | \mathcal{R}(\alpha, \beta, \gamma) | j', m' \rangle = \delta_{jj'} D(j, m, m', \alpha, \beta, \gamma)$$

Where the rotation operator is as defined by the Rotation class [\[R444\]](#) (page 1873).

The Wigner D-function defined in this way gives:

$$D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}$$

Where d is the Wigner small-d function, which is given by `Rotation.d`.

The Wigner small-d function gives the component of the Wigner D-function that is determined by the second Euler angle. That is the Wigner D-function is:

$$D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}$$

Where d is the small-d function. The Wigner D-function is given by `Rotation.D`.

Note that to evaluate the D-function, the j, m and mp parameters must be integer or half integer numbers.

Parameters **j** : Number

Total angular momentum

m : Number

Eigenvalue of angular momentum along axis after rotation

mp : Number

Eigenvalue of angular momentum along rotated axis

alpha : Number, Symbol

First Euler angle of rotation

beta : Number, Symbol

Second Euler angle of rotation

gamma : Number, Symbol

Third Euler angle of rotation

See also:**Rotation** (page 1670) Rotation operator**References**[\[R444\]](#) (page 1873)**Examples**

Evaluate the Wigner-D matrix elements of a simple rotation:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi
>>> rot = Rotation.D(1, 1, 0, pi, pi/2, 0)
>>> rot
WignerD(1, 1, 0, pi, pi/2, 0)
>>> rot.doit()
sqrt(2)/2
```

Evaluate the Wigner-d matrix elements of a simple rotation

```
>>> rot = Rotation.d(1, 1, 0, pi/2)
>>> rot
WignerD(1, 1, 0, 0, pi/2, 0)
>>> rot.doit()
-sqrt(2)/2
```

class sympy.physics.quantum.spin.**JxKet**

Eigenket of Jx.

See JzKet for the usage of spin eigenstates.

See also:**JzKet** (page 1674) Usage of spin states

class sympy.physics.quantum.spin.JxBra

Eigenbra of Jx.

See JzKet for the usage of spin eigenstates.

See also:

[JzKet \(page 1674\)](#) Usage of spin states

class sympy.physics.quantum.spin.JyKet

Eigenket of Jy.

See JzKet for the usage of spin eigenstates.

See also:

[JzKet \(page 1674\)](#) Usage of spin states

class sympy.physics.quantum.spin.JyBra

Eigenbra of Jy.

See JzKet for the usage of spin eigenstates.

See also:

[JzKet \(page 1674\)](#) Usage of spin states

class sympy.physics.quantum.spin.JzKet

Eigenket of Jz.

Spin state which is an eigenstate of the Jz operator. Uncoupled states, that is states representing the interaction of multiple separate spin states, are defined as a tensor product of states.

Parameters **j** : Number, Symbol

Total spin angular momentum

m : Number, Symbol

Eigenvalue of the Jz spin operator

See also:

[JzKetCoupled \(page 1676\)](#) Coupled eigenstates

TensorProduct Used to specify uncoupled states

[uncouple \(page 1679\)](#) Uncouples states given coupling parameters

[couple \(page 1678\)](#) Couples uncoupled states

Examples

Normal States:

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKet, JxBra
>>> from sympy import symbols
>>> JzKet(1, 0)
|1,0>
>>> j, m = symbols('j m')
```

(continues on next page)

(continued from previous page)

```
>>> JzKet(j, m)
|j,m>
```

Rewriting the JzKet in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKet's

```
>>> JzKet(1,1).rewrite("Jx")
|1,-1>/2 - sqrt(2)*|1,0>/2 + |1,1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx, Jz
>>> represent(JzKet(1,-1), basis=Jx)
Matrix([
 [      1/2],
 [sqrt(2)/2],
 [      1/2]])
```

Apply innerproducts between states:

```
>>> from sympy.physics.quantum.innerproduct import InnerProduct
>>> from sympy.physics.quantum.spin import JxBra
>>> i = InnerProduct(JxBra(1,1), JzKet(1,1))
>>> i
<1,1|1,1>
>>> i.doit()
1/2
```

Uncoupled States:

Define an uncoupled state as a TensorProduct between two Jz eigenkets:

```
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> TensorProduct(JzKet(1,0), JzKet(1,1))
|1,0>x|1,1>
>>> TensorProduct(JzKet(j1,m1), JzKet(j2,m2))
|j1,m1>x|j2,m2>
```

A TensorProduct can be rewritten, in which case the eigenstates that make up the tensor product is rewritten to the new basis:

```
>>> TensorProduct(JzKet(1,1),JxBra(1,1)).rewrite('Jz')
|1,1>x|1,-1>/2 + sqrt(2)*|1,1>x|1,0>/2 + |1,1>x|1,1>/2
```

The represent method for TensorProduct's gives the vector representation of the state. Note that the state in the product basis is the equivalent of the tensor product of the vector representation of the component eigenstates:

```
>>> represent(TensorProduct(JzKet(1,0),JzKet(1,1)))
Matrix([
 [0],
 [0],
 [0],
 [1],
 [0],
```

(continues on next page)

(continued from previous page)

```
[0],  
[0],  
[0],  
[0]])  
>>> represent(TensorProduct(JzKet(1,1),JxKet(1,1)), basis=Jz)  
Matrix([[  
[ 1/2],  
[sqrt(2)/2],  
[ 1/2],  
[ 0],  
[ 0],  
[ 0],  
[ 0],  
[ 0],  
[ 0]])
```

class sympy.physics.quantum.spin.JzBra

Eigenbra of Jz.

See the JzKet for the usage of spin eigenstates.

See also:

[JzKet \(page 1674\)](#) Usage of spin states

class sympy.physics.quantum.spin.JxKetCoupled

Coupled eigenket of Jx.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

[JzKetCoupled \(page 1676\)](#) Usage of coupled spin states

class sympy.physics.quantum.spin.JxBraCoupled

Coupled eigenbra of Jx.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

[JzKetCoupled \(page 1676\)](#) Usage of coupled spin states

class sympy.physics.quantum.spin.JyKetCoupled

Coupled eigenket of Jy.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

[JzKetCoupled \(page 1676\)](#) Usage of coupled spin states

class sympy.physics.quantum.spin.JyBraCoupled

Coupled eigenbra of Jy.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

[JzKetCoupled \(page 1676\)](#) Usage of coupled spin states

class sympy.physics.quantum.spin.JzKetCoupled

Coupled eigenket of Jz

Spin state that is an eigenket of Jz which represents the coupling of separate spin spaces.

The arguments for creating instances of JzKetCoupled are j, m, jn and an optional jcoupling argument. The j and m options are the total angular momentum quantum numbers, as used for normal states (e.g. JzKet).

The other required parameter is jn, which is a tuple defining the j_n angular momentum quantum numbers of the product spaces. So for example, if a state represented the coupling of the product basis state $|j_1, m_1\rangle \times |j_2, m_2\rangle$, the jn for this state would be (j1, j2).

The final option is jcoupling, which is used to define how the spaces specified by jn are coupled, which includes both the order these spaces are coupled together and the quantum numbers that arise from these couplings. The jcoupling parameter itself is a list of lists, such that each of the sublists defines a single coupling between the spin spaces. If there are N coupled angular momentum spaces, that is jn has N elements, then there must be N-1 sublists. Each of these sublists making up the jcoupling parameter have length 3. The first two elements are the indices of the product spaces that are considered to be coupled together. For example, if we want to couple j_1 and j_4 , the indices would be 1 and 4. If a state has already been coupled, it is referenced by the smallest index that is coupled, so if j_2 and j_4 has already been coupled to some j_{24} , then this value can be coupled by referencing it with index 2. The final element of the sublist is the quantum number of the coupled state. So putting everything together, into a valid sublist for jcoupling, if j_1 and j_2 are coupled to an angular momentum space with quantum number j_{12} with the value j12, the sublist would be (1,2,j12), N-1 of these sublists are used in the list for jcoupling.

Note the jcoupling parameter is optional, if it is not specified, the default coupling is taken. This default value is to coupled the spaces in order and take the quantum number of the coupling to be the maximum value. For example, if the spin spaces are j_1, j_2, j_3, j_4 , then the default coupling couples j_1 and j_2 to $j_{12} = j_1 + j_2$, then, j_{12} and j_3 are coupled to $j_{123} = j_{12} + j_3$, and finally j_{123} and j_4 to $j = j_{123} + j_4$. The jcoupling value that would correspond to this is:

```
((1,2,j1+j2),(1,3,j1+j2+j3))
```

Parameters args : tuple

The arguments that must be passed are j, m, jn, and jcoupling. The j value is the total angular momentum. The m value is the eigenvalue of the Jz spin operator. The jn list are the j values of angular momentum spaces coupled together. The jcoupling parameter is an optional parameter defining how the spaces are coupled together. See the above description for how these coupling parameters are defined.

See also:

JzKet (page 1674) Normal spin eigenstates

uncouple (page 1679) Uncoupling of coupling spin states

couple (page 1678) Coupling of uncoupled spin states

Examples

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKetCoupled
>>> from sympy import symbols
>>> JzKetCoupled(1, 0, (1, 1))
|1,0,j1=1,j2=1>
>>> j, m, j1, j2 = symbols('j m j1 j2')
>>> JzKetCoupled(j, m, (j1, j2))
|j,m,j1=j1,j2=j2>
```

Defining coupled spin states for more than 2 coupled spaces with various coupling parameters:

```
>>> JzKetCoupled(2, 1, (1, 1, 1))
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((1,2,2),(1,3,2)) )
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((2,3,1),(1,2,2)) )
|2,1,j1=1,j2=1,j3=1,j(2,3)=1>
```

Rewriting the JzKetCoupled in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKetCoupled

```
>>> JzKetCoupled(1,1,(1,1)).rewrite("Jx")
|1,-1,j1=1,j2=1>/2 - sqrt(2)*|1,0,j1=1,j2=1>/2 + |1,1,j1=1,j2=1>/2
```

The rewrite method can be used to convert a coupled state to an uncoupled state. This is done by passing coupled=False to the rewrite function:

```
>>> JzKetCoupled(1, 0, (1, 1)).rewrite('Jz', coupled=False)
-sqrt(2)*|1,-1>x|1,1>/2 + sqrt(2)*|1,1>x|1,-1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx
>>> from sympy import S
>>> represent(JzKetCoupled(1,-1,(S(1)/2,S(1)/2)), basis=Jx)
Matrix([
[      0],
[      1/2],
[sqrt(2)/2],
[      1/2]])
```

class sympy.physics.quantum.spin.JzBraCoupled

Coupled eigenbra of Jz.

See the JzKetCoupled for the usage of coupled spin eigenstates.

See also:

[JzKetCoupled \(page 1676\)](#) Usage of coupled spin states

sympy.physics.quantum.spin.couple(expr, jcoupling_list=None)

Couple a tensor product of spin states

This function can be used to couple an uncoupled tensor product of spin states. All of the eigenstates to be coupled must be of the same class. It will return a linear combination of eigenstates that are subclasses of CoupledSpinState determined by Clebsch-Gordan angular momentum coupling coefficients.

Parameters `expr` : Expr

An expression involving TensorProducts of spin states to be coupled. Each state must be a subclass of SpinState and they all must be the same class.

jcoupling_list : list or tuple

Elements of this list are sub-lists of length 2 specifying the order of the coupling of the spin spaces. The length of this must be N-1, where N is the number of states in the tensor product to be coupled. The elements of this sublist are the same as the first two elements of each sublist in the jcoupling parameter defined for JzKetCoupled. If this parameter is not specified, the default value is taken, which couples the first and second product basis spaces, then couples this new coupled space to the third product space, etc

Examples

Couple a tensor product of numerical states for two spaces:

```
>>> from sympy.physics.quantum.spin import JzKet, couple
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> couple(TensorProduct(JzKet(1,0), JzKet(1,1)))
-sqrt(2)*|1,1,j1=1,j2=1>/2 + sqrt(2)*|2,1,j1=1,j2=1>/2
```

Numerical coupling of three spaces using the default coupling method, i.e. first and second spaces couple, then this couples to the third space:

```
>>> couple(TensorProduct(JzKet(1,1), JzKet(1,1), JzKet(1,0)))
sqrt(6)*|2,2,j1=1,j2=1,j3=1,j(1,2)=2>/3 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,j(1,2)=2>/3
```

Perform this same coupling, but we define the coupling to first couple the first and third spaces:

```
>>> couple(TensorProduct(JzKet(1,1), JzKet(1,1), JzKet(1,0)), ((1,3),(1,2)) )
sqrt(2)*|2,2,j1=1,j2=1,j3=1,j(1,3)=1>/2 - sqrt(6)*|2,2,j1=1,j2=1,j3=1,j(1,3)=2>/6
+ sqrt(3)*|3,2,j1=1,j2=1,j3=1,j(1,3)=2>/3
```

Couple a tensor product of symbolic states:

```
>>> from sympy import symbols
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> couple(TensorProduct(JzKet(j1,m1), JzKet(j2,m2)))
Sum(CG(j1, m1, j2, m2, j, m1 + m2)*|j,m1 + m2,j1=j1,j2=j2>, (j, m1 + m2, j1 + j2))
```

`sympy.physics.quantum.spin.uncouple(expr, jn=None, jcoupling_list=None)`

Uncouple a coupled spin state

Gives the uncoupled representation of a coupled spin state. Arguments must be either a spin state that is a subclass of CoupledSpinState or a spin state that is a subclass of SpinState and an array giving the j values of the spaces that are to be coupled

Parameters `expr` : Expr

The expression containing states that are to be coupled. If the states are a subclass of SpinState, the jn and jcoupling parameters must be defined. If the states are a subclass of CoupledSpinState, jn and jcoupling will be taken from the state.

jn : list or tuple

The list of the j-values that are coupled. If state is a CoupledSpinState, this parameter is ignored. This must be defined if state is not a subclass of CoupledSpinState. The syntax of this parameter is the same as the jn parameter of JzKetCoupled.

jcoupling_list : list or tuple

The list defining how the j-values are coupled together. If state is a CoupledSpinState, this parameter is ignored. This must be defined if state is not a subclass of CoupledSpinState. The syntax of this parameter is the same as the jcoupling parameter of JzKetCoupled.

Examples

Uncouple a numerical state using a CoupledSpinState state:

```
>>> from sympy.physics.quantum.spin import JzKetCoupled, uncouple
>>> from sympy import S
>>> uncouple(JzKetCoupled(1, 0, (S(1)/2, S(1)/2)))
sqrt(2)*|1/2,-1/2>x|1/2,1/2>/2 + sqrt(2)*|1/2,1/2>x|1/2,-1/2>/2
```

Perform the same calculation using a SpinState state:

```
>>> from sympy.physics.quantum.spin import JzKet
>>> uncouple(JzKet(1, 0), (S(1)/2, S(1)/2))
sqrt(2)*|1/2,-1/2>x|1/2,1/2>/2 + sqrt(2)*|1/2,1/2>x|1/2,-1/2>/2
```

Uncouple a numerical state of three coupled spaces using a CoupledSpinState state:

```
>>> uncouple(JzKetCoupled(1, 1, (1, 1, 1), ((1,3,1),(1,2,1)) ))
|1,-1>x|1,1>x|1,1>/2 - |1,0>x|1,0>x|1,1>/2 + |1,1>x|1,0>x|1,0>/2 - |1,1>x|1,1>x|1,
↪ -1>/2
```

Perform the same calculation using a SpinState state:

```
>>> uncouple(JzKet(1, 1), (1, 1, 1), ((1,3,1),(1,2,1)) )
|1,-1>x|1,1>x|1,1>/2 - |1,0>x|1,0>x|1,1>/2 + |1,1>x|1,0>x|1,0>/2 - |1,1>x|1,1>x|1,
↪ -1>/2
```

Uncouple a symbolic state using a CoupledSpinState state:

```
>>> from sympy import symbols
>>> j,m,j1,j2 = symbols('j m j1 j2')
>>> uncouple(JzKetCoupled(j, m, (j1, j2)))
Sum(CG(j1, m1, j2, m2, j, m)*|j1,m1>x|j2,m2>, (m1, -j1, j1), (m2, -j2, j2))
```

Perform the same calculation using a SpinState state

```
>>> uncouple(JzKet(j, m), (j1, j2))
Sum(CG(j1, m1, j2, m2, j, m)*|j1,m1>x|j2,m2>, (m1, -j1, j1), (m2, -j2, j2))
```

State

Dirac notation for states.

class `sympy.physics.quantum.state.KetBase`

Base class for Kets.

This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Ket.

class `sympy.physics.quantum.state.BraBase`

Base class for Bras.

This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Bra.

class `sympy.physics.quantum.state.StateBase`

Abstract base class for general abstract states in quantum mechanics.

All other state classes defined will need to inherit from this class. It carries the basic structure for all other states such as dual, `_eval_adjoint` and `label`.

This is an abstract base class and you should not instantiate it directly, instead use State.

dual

Return the dual state of this one.

classmethod `dual_class()`

Return the class used to construct the dual.

operators

Return the operator(s) that this state is an eigenstate of

class `sympy.physics.quantum.state.State`

General abstract quantum state used as a base class for Ket and Bra.

class `sympy.physics.quantum.state.Ket`

A general time-independent Ket in quantum mechanics.

Inherits from State and KetBase. This class should be used as the base class for all physical, time-independent Kets in a system. This class and its subclasses will be the main classes that users will use for expressing Kets in Dirac notation [R445] (page 1873).

Parameters `args` : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

References

[R445] (page 1873)

Examples

Create a simple Ket and looking at its properties:

```
>>> from sympy.physics.quantum import Ket, Bra
>>> from sympy import symbols, I
>>> k = Ket('psi')
>>> k
|psi>
>>> k.hilbert_space
```

(continues on next page)

(continued from previous page)

```
H
>>> k.is_commutative
False
>>> k.label
(psi,)
```

Ket's know about their associated bra:

```
>>> k.dual
<psi|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.Bra'>
```

Take a linear combination of two kets:

```
>>> k0 = Ket(0)
>>> k1 = Ket(1)
>>> 2*I*k0 - 4*k1
2*I*|0> - 4*|1>
```

Compound labels are passed as tuples:

```
>>> n, m = symbols('n,m')
>>> k = Ket(n,m)
>>> k
|nm>
```

class sympy.physics.quantum.state.Bra

A general time-independent Bra in quantum mechanics.

Inherits from State and BraBase. A Bra is the dual of a Ket [\[R446\]](#) (page 1873). This class and its subclasses will be the main classes that users will use for expressing Bras in Dirac notation.

Parameters args : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

References

[\[R446\]](#) (page 1873)

Examples

Create a simple Bra and look at its properties:

```
>>> from sympy.physics.quantum import Ket, Bra
>>> from sympy import symbols, I
>>> b = Bra('psi')
>>> b
<psi|
>>> b.hilbert_space
```

(continues on next page)

(continued from previous page)

```
H
>>> b.is_commutative
False
```

Bra's know about their dual Ket's:

```
>>> b.dual
|psi>
>>> b.dual_class()
<class 'sympy.physics.quantum.state.Ket'>
```

Like Kets, Bras can have compound labels and be manipulated in a similar manner:

```
>>> n, m = symbols('n,m')
>>> b = Bra(n,m) - I*Bra(m,n)
>>> b
-I*<mn| + <nm|
```

Symbols in a Bra can be substituted using `.subs`:

```
>>> b.subs(n,m)
<mm| - I*<mm|
```

class sympy.physics.quantum.state.TimeDepState

Base class for a general time-dependent quantum state.

This class is used as a base class for any time-dependent state. The main difference between this class and the time-independent state is that this class takes a second argument that is the time in addition to the usual label argument.

Parameters `args` : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

label

The label of the state.

time

The time of the state.

class sympy.physics.quantum.state.TimeDepBra

General time-dependent Bra in quantum mechanics.

This inherits from TimeDepState and BraBase and is the main class that should be used for Bras that vary with time. Its dual is a TimeDepKet.

Parameters `args` : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

Examples

```
>>> from sympy.physics.quantum import TimeDepBra
>>> from sympy import symbols, I
>>> b = TimeDepBra('psi', 't')
>>> b
<psi;t|
>>> b.time
t
>>> b.label
(psi,)
>>> b.hilbert_space
H
>>> b.dual
|psi;t>
```

class sympy.physics.quantum.state.TimeDepKet

General time-dependent Ket in quantum mechanics.

This inherits from TimeDepState and KetBase and is the main class that should be used for Kets that vary with time. Its dual is a TimeDepBra.

Parameters *args* : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

Examples

Create a TimeDepKet and look at its attributes:

```
>>> from sympy.physics.quantum import TimeDepKet
>>> k = TimeDepKet('psi', 't')
>>> k
|psi;t>
>>> k.time
t
>>> k.label
(psi,)
>>> k.hilbert_space
H
```

TimeDepKets know about their dual bra:

```
>>> k.dual
<psi;t|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.TimeDepBra'>
```

class sympy.physics.quantum.state.Wavefunction

Class for representations in continuous bases

This class takes an expression and coordinates in its constructor. It can be used to easily calculate normalizations and probabilities.

Parameters *expr* : Expr

The expression representing the functional form of the w.f.

coords : Symbol or tuple

The coordinates to be integrated over, and their bounds

Examples

Particle in a box, specifying bounds in the more primitive way of using Piecewise:

```
>>> from sympy import Symbol, Piecewise, pi, N
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = Symbol('x', real=True)
>>> n = 1
>>> L = 1
>>> g = Piecewise((0, x < 0), (0, x > L), (sqrt(2//L)*sin(n*pi*x/L), True))
>>> f = Wavefunction(g, x)
>>> f.norm
1
>>> f.is_normalized
True
>>> p = f.prob()
>>> p(0)
0
>>> p(L)
0
>>> p(0.5)
2
>>> p(0.85*L)
2*sin(0.85*pi)**2
>>> N(p(0.85*L))
0.412214747707527
```

Additionally, you can specify the bounds of the function and the indices in a more compact way:

```
>>> from sympy import symbols, pi, diff
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> f(L+1)
0
>>> f(L-1)
sqrt(2)*sin(pi*n*(L - 1)/L)/sqrt(L)
>>> f(-1)
0
>>> f(0.85)
sqrt(2)*sin(0.85*pi*n/L)/sqrt(L)
>>> f(0.85, n=1, L=1)
sqrt(2)*sin(0.85*pi)
>>> f.is_commutative
False
```

All arguments are automatically sympified, so you can define the variables as strings rather than symbols:

```
>>> expr = x**2
>>> f = Wavefunction(expr, 'x')
>>> type(f.variables[0])
<class 'sympy.core.symbol.Symbol'>
```

Derivatives of Wavefunctions will return Wavefunctions:

```
>>> diff(f, x)
Wavefunction(2*x, x)
```

expr

Return the expression which is the functional form of the Wavefunction

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, x)
>>> f.expr
x**2
```

is_commutative

Override Function's `is_commutative` so that order is preserved in represented expressions

is_normalized

Returns true if the Wavefunction is properly normalized

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x, L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.is_normalized
True
```

limits

Return the limits of the coordinates which the w.f. depends on If no limits are specified, defaults to $(-\infty, \infty)$.

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, (x, 0, 1))
```

(continues on next page)

(continued from previous page)

```

>>> f.limits
{x: (0, 1)}
>>> f = Wavefunction(x**2, x)
>>> f.limits
{x: (-oo, oo)}
>>> f = Wavefunction(x**2 + y**2, x, (y, -1, 2))
>>> f.limits
{x: (-oo, oo), y: (-1, 2)}

```

norm

Return the normalization of the specified functional form.

This function integrates over the coordinates of the Wavefunction, with the bounds specified.

Examples

```

>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
sqrt(2)*sqrt(L)/2

```

normalize()

Return a normalized version of the Wavefunction

Examples

```

>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = symbols('x', real=True)
>>> L = symbols('L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.normalize()
Wavefunction(sqrt(2)*sin(pi*n*x/L)/sqrt(L), (x, 0, L))

```

prob()

Return the absolute magnitude of the w.f., $|\psi(x)|^2$

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', real=True)
>>> n = symbols('n', integer=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.prob()
Wavefunction(sin(pi*n*x/L)**2, x)
```

variables

Return the coordinates which the wavefunction depends on

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x,y = symbols('x,y')
>>> f = Wavefunction(x*y, x, y)
>>> f.variables
(x, y)
>>> g = Wavefunction(x*y, x)
>>> g.variables
(x,)
```

Quantum Computation

Circuit Plot

Matplotlib based plotting of quantum circuits.

Todo:

- Optimize printing of large circuits.
- Get this to work with single gates.
- Do a better job checking the form of circuits to make sure it is a Mul of Gates.
- Get multi-target gates plotting.
- Get initial and final states to plot.
- Get measurements to plot. Might need to rethink measurement as a gate issue.
- Get scale and figsize to be handled in a better way.
- Write some tests/examples!

`sympy.physics.quantum.circuitplot.labeller(n, symbol='q')`

Autogenerate labels for wires of quantum circuits.

Parameters **n** : int

number of qubits in the circuit

symbol : string

A character string to precede all gate labels. E.g. 'q_0', 'q_1', etc.

```
>>> from sympy.physics.quantum.circuitplot import labeller
>>> labeller(2)
['q_1', 'q_0']
>>> labeller(3,'j')
['j_2', 'j_1', 'j_0']
```

class sympy.physics.quantum.circuitplot.Mz

Mock-up of a z measurement gate.

This is in circuitplot rather than gate.py because it's not a real gate, it just draws one.

class sympy.physics.quantum.circuitplot.Mx

Mock-up of an x measurement gate.

This is in circuitplot rather than gate.py because it's not a real gate, it just draws one.

sympy.physics.quantum.circuitplot.CreateCGate(name, latexname=None)

Use a lexical closure to make a controlled gate.

Gates

An implementation of gates that act on qubits.

Gates are unitary operators that act on the space of qubits.

Medium Term Todo:

- Optimize Gate._apply_operators_Qubit to remove the creation of many intermediate Qubit objects.
- Add commutation relationships to all operators and use this in gate_sort.
- Fix gate_sort and gate_simp.
- Get multi-target UGates plotting properly.
- Get UGate to work with either sympy/numpy matrices and output either format. This should also use the matrix slots.

class sympy.physics.quantum.gate.Gate

Non-controlled unitary gate operator that acts on qubits.

This is a general abstract gate that needs to be subclassed to do anything useful.

Parameters label : tuple, int

A list of the target qubits (as ints) that the gate will apply to.

get_target_matrix(format='sympy')

The matrix rep. of the target part of the gate.

Parameters format : str

The format string ('sympy', 'numpy', etc.)

min_qubits

The minimum number of qubits this gate needs to act on.

nqubits

The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

targets

A tuple of target qubits.

class sympy.physics.quantum.gate.**CGate**

A general unitary gate with control qubits.

A general control gate applies a target gate to a set of targets if all of the control qubits have a particular values (set by `CGate.control_value`).

Parameters label : tuple

The label in this case has the form (controls, gate), where controls is a tuple/list of control qubits (as ints) and gate is a Gate instance that is the target operator.

controls

A tuple of control qubits.

decompose(**options)

Decompose the controlled gate into CNOT and single qubits gates.

eval_controls(qubit)

Return True/False to indicate if the controls are satisfied.

gate

The non-controlled gate that will be applied to the targets.

min_qubits

The minimum number of qubits this gate needs to act on.

nqubits

The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

plot_gate(circ_plot, gate_idx)

Plot the controlled gate. If `simplify_cgate` is true, simplify C-X and C-Z gates into their more familiar forms.

targets

A tuple of target qubits.

class sympy.physics.quantum.gate.**UGate**

General gate specified by a set of targets and a target matrix.

Parameters label : tuple

A tuple of the form (targets, U), where targets is a tuple of the target qubits and U is a unitary matrix with dimension of `len(targets)`.

get_target_matrix(format='sympy')

The matrix rep. of the target part of the gate.

Parameters format : str

The format string ('sympy', 'numpy', etc.)

targets

A tuple of target qubits.

class sympy.physics.quantum.gate.**OneQubitGate**

A single qubit unitary gate base class.

class sympy.physics.quantum.gate.**TwoQubitGate**

A two qubit unitary gate base class.

class sympy.physics.quantum.gate.IdentityGate

The single qubit identity gate.

Parameters target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.HadamardGate

The single qubit Hadamard gate.

Parameters target : int

The target qubit this gate will apply to.

Examples

```
>>> from sympy import sqrt
>>> from sympy.physics.quantum.qubit import Qubit
>>> from sympy.physics.quantum.gate import HadamardGate
>>> from sympy.physics.quantum.qapply import qapply
>>> qapply(HadamardGate(0)*Qubit('1'))
sqrt(2)*|0>/2 - sqrt(2)*|1>/2
>>> # Hadamard on bell state, applied on 2 qubits.
>>> psi = 1/sqrt(2)*(Qubit('00')+Qubit('11'))
>>> qapply(HadamardGate(0)*HadamardGate(1)*psi)
sqrt(2)*|00>/2 + sqrt(2)*|11>/2
```

class sympy.physics.quantum.gate.XGate

The single qubit X, or NOT, gate.

Parameters target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.YGate

The single qubit Y gate.

Parameters target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.ZGate

The single qubit Z gate.

Parameters target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.TGate

The single qubit $\pi/8$ gate.

This gate rotates the phase of the state by $\pi/4$ if the state is $|1\rangle$ and does nothing if the state is $|0\rangle$.

Parameters target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.PhaseGate

The single qubit phase, or S, gate.

This gate rotates the phase of the state by $\pi/2$ if the state is $|1\rangle$ and does nothing if the state is $|0\rangle$.

Parameters target : int

The target qubit this gate will apply to.

class `sympy.physics.quantum.gate.SwapGate`

Two qubit SWAP gate.

This gate swap the values of the two qubits.

Parameters **label** : tuple

A tuple of the form (target1, target2).

decompose(**options)

Decompose the SWAP gate into CNOT gates.

class `sympy.physics.quantum.gate.CNotGate`

Two qubit controlled-NOT.

This gate performs the NOT or X gate on the target qubit if the control qubits all have the value 1.

Parameters **label** : tuple

A tuple of the form (control, target).

Examples

```
>>> from sympy.physics.quantum.gate import CNOT
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import Qubit
>>> c = CNOT(1,0)
>>> qapply(c*Qubit('10')) # note that qubits are indexed from right to left
|11>
```

controls

A tuple of control qubits.

gate

The non-controlled gate that will be applied to the targets.

min_qubits

The minimum number of qubits this gate needs to act on.

targets

A tuple of target qubits.

`sympy.physics.quantum.gate.CNOT`

alias of `sympy.physics.quantum.gate.CNotGate` (page 1692)

`sympy.physics.quantum.gate.SWAP`

alias of `sympy.physics.quantum.gate.SwapGate` (page 1692)

`sympy.physics.quantum.gate.H`

alias of `sympy.physics.quantum.gate.HadamardGate` (page 1691)

`sympy.physics.quantum.gate.X`

alias of `sympy.physics.quantum.gate.XGate` (page 1691)

`sympy.physics.quantum.gate.Y`

alias of `sympy.physics.quantum.gate.YGate` (page 1691)

`sympy.physics.quantum.gate.Z`

alias of `sympy.physics.quantum.gate.ZGate` (page 1691)

`sympy.physics.quantum.gate.T`

alias of `sympy.physics.quantum.gate.TGate` (page 1691)

`sympy.physics.quantum.gate.S`

alias of `sympy.physics.quantum.gate.PhaseGate` (page 1691)

`sympy.physics.quantum.gate.Phase`

alias of `sympy.physics.quantum.gate.PhaseGate` (page 1691)

`sympy.physics.quantum.gate.normalized(normalize)`

Set flag controlling normalization of Hadamard gates by $1/\sqrt{2}$.

This is a global setting that can be used to simplify the look of various expressions, by leaving off the leading $1/\sqrt{2}$ of the Hadamard gate.

Parameters `normalize` : bool

Should the Hadamard gate include the $1/\sqrt{2}$ normalization factor?
When True, the Hadamard gate will have the $1/\sqrt{2}$. When False, the Hadamard gate will not have this factor.

`sympy.physics.quantum.gate.gate_sort(circuit)`

Sorts the gates while keeping track of commutation relations

This function uses a bubble sort to rearrange the order of gate application. Keeps track of Quantum computations special commutation relations (e.g. things that apply to the same Qubit do not commute with each other)

circuit is the Mul of gates that are to be sorted.

`sympy.physics.quantum.gate.gate_simp(circuit)`

Simplifies gates symbolically

It first sorts gates using `gate_sort`. It then applies basic simplification rules to the circuit, e.g., $XGate^{**2} = Identity$

`sympy.physics.quantum.gate.random_circuit(ngates, nqubits, gate_space=(<class 'sympy.physics.quantum.gate.XGate'>,
<class 'sympy.physics.quantum.gate.YGate'>,
<class 'sympy.physics.quantum.gate.ZGate'>,
<class 'sympy.physics.quantum.gate.PhaseGate'>,
<class 'sympy.physics.quantum.gate.TGate'>,
<class 'sympy.physics.quantum.gate.HadamardGate'>,
<class 'sympy.physics.quantum.gate.CNotGate'>,
<class 'sympy.physics.quantum.gate.SwapGate'>))`

Return a random circuit of `ngates` and `nqubits`.

This uses an equally weighted sample of (X, Y, Z, S, T, H, CNOT, SWAP) gates.

Parameters `ngates` : int

The number of gates in the circuit.

nqubits : int

The number of qubits in the circuit.

gate_space : tuple

A tuple of the gate classes that will be used in the circuit. Repeating gate classes multiple times in this tuple will increase the frequency they appear in the random circuit.

class `sympy.physics.quantum.gate.CGates`

Version of CGate that allows gate simplifications. I.e. `cnot` looks like an `oplus`, `cphase` has dots, etc.

Grover's Algorithm

Grover's algorithm and helper functions.

Todo:

- W gate construction (or perhaps -W gate based on Mermin's book)
- Generalize the algorithm for an unknown function that returns 1 on multiple qubit states, not just one.
- Implement `_represent_ZGate` in `OracleGate`

class `sympy.physics.quantum.grover.OracleGate`

A black box gate.

The gate marks the desired qubits of an unknown function by flipping the sign of the qubits. The unknown function returns true when it finds its desired qubits and false otherwise.

Parameters `qubits` : int

Number of qubits.

oracle : callable

A callable function that returns a boolean on a computational basis.

Examples

Apply an Oracle gate that flips the sign of $|2\rangle$ on different qubits:

```
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.grover import OracleGate
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(2, f)
>>> qapply(v*IntQubit(2))
- |2>
>>> qapply(v*IntQubit(3))
|3>
```

search_function

The unknown function that helps find the sought after qubits.

targets

A tuple of target qubits.

class `sympy.physics.quantum.grover.WGate`

General n qubit W Gate in Grover's algorithm.

The gate performs the operation $2|\phi\rangle\langle\phi| - 1$ on some qubits. $|\phi\rangle = (\text{tensor product of } n \text{ Hadamards}) * (|0\rangle \text{ with } n \text{ qubits})$

Parameters `nqubits` : int

The number of qubits to operate on

`sympy.physics.quantum.grover.superposition_basis(nqubits)`

Creates an equal superposition of the computational basis.

Parameters `nqubits` : int

The number of qubits.

Returns state : Qubit

An equal superposition of the computational basis with nqubits.

Examples

Create an equal superposition of 2 qubits:

```
>>> from sympy.physics.quantum.grover import superposition_basis
>>> superposition_basis(2)
|0>/2 + |1>/2 + |2>/2 + |3>/2
```

`sympy.physics.quantum.grover.grover_iteration(qstate, oracle)`

Applies one application of the Oracle and W Gate, WV.

Parameters qstate : Qubit

A superposition of qubits.

oracle : OracleGate

The black box operator that flips the sign of the desired basis qubits.

Returns Qubit : The qubits after applying the Oracle and W gate.

Examples

Perform one iteration of grover's algorithm to see a phase change:

```
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import OracleGate
>>> from sympy.physics.quantum.grover import superposition_basis
>>> from sympy.physics.quantum.grover import grover_iteration
>>> numqubits = 2
>>> basis_states = superposition_basis(numqubits)
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(numqubits, f)
>>> qapply(grover_iteration(basis_states, v))
|2>
```

`sympy.physics.quantum.grover.apply_grover(oracle, nqubits, iterations=None)`

Applies grover's algorithm.

Parameters oracle : callable

The unknown callable function that returns true when applied to the desired qubits and false otherwise.

Returns state : Expr

The resulting state after Grover's algorithm has been iterated.

Examples

Apply grover's algorithm to an even superposition of 2 qubits:

```
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import apply_grover
>>> f = lambda qubits: qubits == IntQubit(2)
>>> qapply(apply_grover(f, 2))
|2>
```

QFT

An implementation of qubits and gates acting on them.

Todo:

- Update docstrings.
- Update tests.
- Implement apply using decompose.
- Implement represent using decompose or something smarter. For this to work we first have to implement represent for SWAP.
- Decide if we want upper index to be inclusive in the constructor.
- Fix the printing of R_k gates in plotting.

class sympy.physics.quantum.qft.QFT

The forward quantum Fourier transform.

decompose()

Decomposes QFT into elementary gates.

class sympy.physics.quantum.qft.IQFT

The inverse quantum Fourier transform.

decompose()

Decomposes IQFT into elementary gates.

class sympy.physics.quantum.qft.RkGate

This is the R_k gate of the QFT.

sympy.physics.quantum.qft.Rk

alias of [sympy.physics.quantum.qft.RkGate](#) (page 1696)

Qubit

Qubits for quantum computing.

Todo: * Finish implementing measurement logic. This should include POVM. * Update docstrings. * Update tests.

class sympy.physics.quantum.qubit.Qubit

A multi-qubit ket in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so $|00001\rangle$ has a 1 in the least significant qubit.

Parameters values : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

Examples

Create a qubit in a couple of different ways and look at their attributes:

```
>>> from sympy.physics.quantum.qubit import Qubit
>>> Qubit(0,0,0)
|000>
>>> q = Qubit('0101')
>>> q
|0101>
```

```
>>> q.nqubits
4
>>> len(q)
4
>>> q.dimension
4
>>> q.qubit_values
(0, 1, 0, 1)
```

We can flip the value of an individual qubit:

```
>>> q.flip(1)
|0111>
```

We can take the dagger of a Qubit to get a bra:

```
>>> from sympy.physics.quantum.dagger import Dagger
>>> Dagger(q)
<0101|
>>> type(Dagger(q))
<class 'sympy.physics.quantum.qubit.QubitBra'>
```

Inner products work as expected:

```
>>> ip = Dagger(q)*q
>>> ip
<0101|0101>
>>> ip.doit()
1
```

class sympy.physics.quantum.qubit.QubitBra

A multi-qubit bra in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so $|00001\rangle$ has a 1 in the least significant qubit.

Parameters values : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

See also:

Qubit (page 1696) Examples using qubits

class sympy.physics.quantum.qubit.IntQubit

A qubit ket that store integers as binary numbers in qubit values.

The differences between this class and Qubit are:

- The form of the constructor.

- The qubit values are printed as their corresponding integer, rather than the raw qubit values. The internal storage format of the qubit values in the same as Qubit.

Parameters values : int, tuple

If a single argument, the integer we want to represent in the qubit values. This integer will be represented using the fewest possible number of qubits. If a pair of integers, the first integer gives the integer to represent in binary form and the second integer gives the number of qubits to use.

Examples

Create a qubit for the integer 5:

```
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qubit import Qubit
>>> q = IntQubit(5)
>>> q
|5>
```

We can also create an IntQubit by passing a Qubit instance.

```
>>> q = IntQubit(Qubit('101'))
>>> q
|5>
>>> q.as_int()
5
>>> q.nqubits
3
>>> q.qubit_values
(1, 0, 1)
```

We can go back to the regular qubit form.

```
>>> Qubit(q)
|101>
```

class sympy.physics.quantum.qubit.IntQubitBra

A qubit bra that store integers as binary numbers in qubit values.

sympy.physics.quantum.qubit.qubit_to_matrix(qubit, format='sympy')

Converts an Add/Mul of Qubit objects into it's matrix representation

This function is the inverse of matrix_to_qubit and is a shorthand for represent(qubit).

sympy.physics.quantum.qubit.matrix_to_qubit(matrix)

Convert from the matrix repr. to a sum of Qubit objects.

Parameters matrix : Matrix, numpy.matrix, scipy.sparse

The matrix to build the Qubit representation of. This works with sympy matrices, numpy matrices and scipy.sparse sparse matrices.

Examples

Represent a state and then go back to its qubit form:


```
>>> from sympy.physics.quantum.qubit import matrix_to_qubit, Qubit
>>> from sympy.physics.quantum.gate import Z
>>> from sympy.physics.quantum.represent import represent
>>> q = Qubit('01')
>>> matrix_to_qubit(represent(q))
|01>
```

`sympy.physics.quantum.qubit.matrix_to_density(mat)`

Works by finding the eigenvectors and eigenvalues of the matrix. We know we can decompose rho by doing: `sum(EigenVal*|Eigenvect><Eigenvect|)`

`sympy.physics.quantum.qubit.measure_all(qubit, format='sympy', normalize=True)`

Perform an ensemble measurement of all qubits.

Parameters `qubit` : Qubit, Add

The qubit to measure. This can be any Qubit or a linear combination of them.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

Returns `result` : list

A list that consists of primitive states and their probabilities.

Examples

```
>>> from sympy.physics.quantum.qubit import Qubit, measure_all
>>> from sympy.physics.quantum.gate import H, X, Y, Z
>>> from sympy.physics.quantum.qapply import qapply
```

```
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_all(q)
[(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]
```

`sympy.physics.quantum.qubit.measure_partial(qubit, bits, format='sympy', normalize=True)`

Perform a partial ensemble measure on the specified qubits.

Parameters `qubits` : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

bits : tuple

The qubits to measure.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

Returns result : list

A list that consists of primitive states and their probabilities.

Examples

```
>>> from sympy.physics.quantum.qubit import Qubit, measure_partial
>>> from sympy.physics.quantum.gate import H, X, Y, Z
>>> from sympy.physics.quantum.qapply import qapply
```

```
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_partial(q, (0,))
[(sqrt(2)*|00>/2 + sqrt(2)*|10>/2, 1/2), (sqrt(2)*|01>/2 + sqrt(2)*|11>/2, 1/2)]
```

`sympy.physics.quantum.qubit.measure_partial_oneshot(qubit, bits, format='sympy')`

Perform a partial oneshot measurement on the specified qubits.

A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns one of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

Parameters qubits : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

bits : tuple

The qubits to measure.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

Returns result : Qubit

The qubit that the system collapsed to upon measurement.

`sympy.physics.quantum.qubit.measure_all_oneshot(qubit, format='sympy')`

Perform a oneshot ensemble measurement on all qubits.

A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns one of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

Parameters qubits : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

Returns result : Qubit

The qubit that the system collapsed to upon measurement.

Shor's Algorithm

Shor's algorithm and helper functions.

Todo:

- Get the CMod gate working again using the new Gate API.
- Fix everything.
- Update docstrings and reformat.
- Remove print statements. We may want to think about a better API for this.

class sympy.physics.quantum.shor.CMod

A controlled mod gate.

This is black box controlled Mod function for use by shor's algorithm. TODO implement a decompose property that returns how to do this in terms of elementary gates

N

N is the type of modular arithmetic we are doing.

a

Base of the controlled mod function.

t

Size of 1/2 input register. First 1/2 holds output.

sympy.physics.quantum.shor.**period_find**(a, N)

Finds the period of a in modulo N arithmetic

This is quantum part of Shor's algorithm. It takes two registers, puts first in superposition of states with Hadamards so: $|k\rangle|0\rangle$ with k being all possible choices. It then does a controlled mod and a QFT to determine the order of a.

sympy.physics.quantum.shor.**shor**(N)

This function implements Shor's factoring algorithm on the Integer N

The algorithm starts by picking a random number (a) and seeing if it is coprime with N. If it isn't, then the gcd of the two numbers is a factor and we are done. Otherwise, it begins the period_finding subroutine which finds the period of a in modulo N arithmetic. This period, if even, can be used to calculate factors by taking $a^{(r/2)-1}$ and $a^{(r/2)+1}$. These values are returned.

Analytic Solutions

Particle in a Box

1D quantum particle in a box.

class sympy.physics.quantum.piab.PIABHamiltonian

Particle in a box Hamiltonian operator.

class sympy.physics.quantum.piab.PIABKet
Particle in a box eigenket.

class sympy.physics.quantum.piab.PIABBra
Particle in a box eigenbra.

Optics Module

Abstract

Contains docstrings of Physics-Optics module

Gaussian Optics

Gaussian optics.

The module implements:

- Ray transfer matrices for geometrical and gaussian optics.
See RayTransferMatrix, GeometricRay and BeamParameter
- Conjugation relations for geometrical and gaussian optics.
See geometric_conj*, gauss_conj and conjugate_gauss_beams

The conventions for the distances are as follows:

focal distance positive for convergent lenses

object distance positive for real objects

image distance positive for real images

class sympy.physics.optics.gaussopt.RayTransferMatrix
Base class for a Ray Transfer Matrix.

It should be used if there isn't already a more specific subclass mentioned in See Also.

Parameters **parameters** : A, B, C and D or 2x2 matrix (Matrix(2, 2, [A, B, C, D]))

See also:

[GeometricRay](#) (page 1707), [BeamParameter](#) (page 1708), [FreeSpace](#) (page 1704), [FlatRefraction](#) (page 1704), [CurvedRefraction](#) (page 1705), [FlatMirror](#) (page 1705), [CurvedMirror](#) (page 1706), [ThinLens](#) (page 1706)

References

[R425] (page 1873)

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix, ThinLens
>>> from sympy import Symbol, Matrix
```

```
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat
Matrix([
[1, 2],
[3, 4]])
```

```
>>> RayTransferMatrix(Matrix([[1, 2], [3, 4]]))
Matrix([
[1, 2],
[3, 4]])
```

```
>>> mat.A
1
```

```
>>> f = Symbol('f')
>>> lens = ThinLens(f)
>>> lens
Matrix([
[ 1, 0],
[-1/f, 1]])
```

```
>>> lens.C
-1/f
```

Attributes

cols	
rows	

A

The A parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.A
1
```

B

The B parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.B
2
```

C

The C parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.C
3
```

D

The D parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.D
4
```

class sympy.physics.optics.gaussopt.**FreeSpace**

Ray Transfer Matrix for free space.

Parameters distance

See also:

[RayTransferMatrix](#) (page 1702)

Examples

```
>>> from sympy.physics.optics import FreeSpace
>>> from sympy import symbols
>>> d = symbols('d')
>>> FreeSpace(d)
Matrix([
[1, d],
[0, 1]])
```

Attributes

cols	
rows	

class sympy.physics.optics.gaussopt.**FlatRefraction**

Ray Transfer Matrix for refraction.

Parameters n1 : refractive index of one medium

n2 : refractive index of other medium

See also:[RayTransferMatrix](#) (page 1702)**Examples**

```
>>> from sympy.physics.optics import FlatRefraction
>>> from sympy import symbols
>>> n1, n2 = symbols('n1 n2')
>>> FlatRefraction(n1, n2)
Matrix([
[1,      0],
[0, n1/n2]])
```

Attributes

cols	
rows	

class sympy.physics.optics.gaussopt.**CurvedRefraction**

Ray Transfer Matrix for refraction on curved interface.

Parameters **R** : radius of curvature (positive for concave)

n1 : refractive index of one medium

n2 : refractive index of other medium

See also:[RayTransferMatrix](#) (page 1702)**Examples**

```
>>> from sympy.physics.optics import CurvedRefraction
>>> from sympy import symbols
>>> R, n1, n2 = symbols('R n1 n2')
>>> CurvedRefraction(R, n1, n2)
Matrix([
[      1,      0],
[(n1 - n2)/(R*n2), n1/n2]])
```

Attributes

cols	
rows	

class sympy.physics.optics.gaussopt.**FlatMirror**

Ray Transfer Matrix for reflection.

See also:

[RayTransferMatrix](#) (page 1702)

Examples

```
>>> from sympy.physics.optics import FlatMirror
>>> FlatMirror()
Matrix([
[1, 0],
[0, 1]])
```

Attributes

cols	
rows	

class sympy.physics.optics.gaussopt.**CurvedMirror**
Ray Transfer Matrix for reflection from curved surface.

Parameters **R** : radius of curvature (positive for concave)

See also:

[RayTransferMatrix](#) (page 1702)

Examples

```
>>> from sympy.physics.optics import CurvedMirror
>>> from sympy import symbols
>>> R = symbols('R')
>>> CurvedMirror(R)
Matrix([
[ 1, 0],
[-2/R, 1]])
```

Attributes

cols	
rows	

class sympy.physics.optics.gaussopt.**ThinLens**
Ray Transfer Matrix for a thin lens.

Parameters **f** : the focal distance

See also:

[RayTransferMatrix](#) (page 1702)

Examples

```
>>> from sympy.physics.optics import ThinLens
>>> from sympy import symbols
>>> f = symbols('f')
>>> ThinLens(f)
Matrix([
[ 1, 0],
[-1/f, 1]])
```

Attributes

cols	
rows	

class sympy.physics.optics.gaussopt.**GeometricRay**

Representation for a geometric ray in the Ray Transfer Matrix formalism.

Parameters **h** : height, and

angle : angle, or

matrix : a 2x1 matrix (Matrix(2, 1, [height, angle]))

See also:

[RayTransferMatrix](#) (page 1702)

Examples

```
>>> from sympy.physics.optics import GeometricRay, FreeSpace
>>> from sympy import symbols, Matrix
>>> d, h, angle = symbols('d, h, angle')
```

```
>>> GeometricRay(h, angle)
Matrix([
[ h],
[angle]])
```

```
>>> FreeSpace(d)*GeometricRay(h, angle)
Matrix([
[angle*d + h],
[ angle]])
```

```
>>> GeometricRay( Matrix( ((h,), (angle,)) ) )
Matrix([
[ h],
[angle]])
```

Attributes

cols	
rows	

angle

The angle with the optical axis.

Examples

```
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.angle
angle
```

height

The distance from the optical axis.

Examples

```
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.height
h
```

class `sympy.physics.optics.gaussopt.BeamParameter`

Representation for a gaussian ray in the Ray Transfer Matrix formalism.

Parameters **wavelen** : the wavelength,

z : the distance to waist, and

w : the waist, or

z_r : the rayleigh range

See also:

[RayTransferMatrix](#) (page 1702)

References

[R426] (page 1873), [R427] (page 1873)

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

```
>>> p.q.n()
1.0 + 5.92753330865999*I
>>> p.w_0.n()
0.0010000000000000000
>>> p.z_r.n()
5.92753330865999
```

```
>>> from sympy.physics.optics import FreeSpace
>>> fs = FreeSpace(10)
>>> p1 = fs*p
>>> p.w.n()
0.00101413072159615
>>> p1.w.n()
0.00210803120913829
```

divergence

Half of the total angular spread.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.divergence
0.00053/pi
```

gouy

The Gouy phase.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.gouy
atan(0.53/pi)
```

q

The complex parameter representing the beam.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

radius

The radius of curvature of the phase front.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.radius
1 + 3.55998576005696*pi**2
```

w

The beam radius at $1/e^2$ intensity.

See also:

w_0 (page 1710) the minimal radius of beam

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w
0.001*sqrt(0.2809/pi**2 + 1)
```

w_0

The beam waist (minimal radius).

See also:

w (page 1710) the beam radius at $1/e^2$ intensity

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w_0
0.001000000000000000
```

waist_approximation_limit

The minimal waist for which the gauss beam approximation is valid.

The gauss beam is a solution to the paraxial equation. For curvatures that are too great it is not a valid approximation.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.waist_approximation_limit
1.06e-6/pi
```

`sympy.physics.optics.gaussopt.waist2rayleigh(w, wavelen)`

Calculate the rayleigh range from the waist of a gaussian beam.

See also:

`rayleigh2waist` (page 1711), `BeamParameter` (page 1708)

Examples

```
>>> from sympy.physics.optics import waist2rayleigh
>>> from sympy import symbols
>>> w, wavelen = symbols('w wavelen')
>>> waist2rayleigh(w, wavelen)
pi*w**2/wavelen
```

`sympy.physics.optics.gaussopt.rayleigh2waist(z_r, wavelen)`
Calculate the waist from the rayleigh range of a gaussian beam.

See also:

[waist2rayleigh](#) (page 1710), [BeamParameter](#) (page 1708)

Examples

```
>>> from sympy.physics.optics import rayleigh2waist
>>> from sympy import symbols
>>> z_r, wavelen = symbols('z_r wavelen')
>>> rayleigh2waist(z_r, wavelen)
sqrt(wavelen*z_r)/sqrt(pi)
```

`sympy.physics.optics.gaussopt.geometric_conj_ab(a, b)`
Conjugation relation for geometrical beams under paraxial conditions.
Takes the distances to the optical element and returns the needed focal distance.

See also:

[geometric_conj_af](#) (page 1711), [geometric_conj_bf](#) (page 1712)

Examples

```
>>> from sympy.physics.optics import geometric_conj_ab
>>> from sympy import symbols
>>> a, b = symbols('a b')
>>> geometric_conj_ab(a, b)
a*b/(a + b)
```

`sympy.physics.optics.gaussopt.geometric_conj_af(a, f)`
Conjugation relation for geometrical beams under paraxial conditions.
Takes the object distance (for `geometric_conj_af`) or the image distance (for `geometric_conj_bf`) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

See also:

[geometric_conj_ab](#) (page 1711)

Examples

```
>>> from sympy.physics.optics.gaussopt import geometric_conj_af, geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

`sympy.physics.optics.gaussopt.geometric_conj_bf(a, f)`

Conjugation relation for geometrical beams under paraxial conditions.

Takes the object distance (for `geometric_conj_af`) or the image distance (for `geometric_conj_bf`) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

See also:

`geometric_conj_ab` (page 1711)

Examples

```
>>> from sympy.physics.optics.gaussopt import geometric_conj_af, geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

`sympy.physics.optics.gaussopt.gaussian_conj(s_in, z_r_in, f)`

Conjugation relation for gaussian beams.

Parameters `s_in` : the distance to optical element from the waist

`z_r_in` : the rayleigh range of the incident beam

`f` : the focal length of the optical element

Returns a tuple containing (`s_out`, `z_r_out`, `m`)

`s_out` : the distance between the new waist and the optical element

`z_r_out` : the rayleigh range of the emergent beam

`m` : the ration between the new and the old waists

Examples

```
>>> from sympy.physics.optics import gaussian_conj
>>> from sympy import symbols
>>> s_in, z_r_in, f = symbols('s_in z_r_in f')
```

```
>>> gaussian_conj(s_in, z_r_in, f)[0]
1/(-1/(s_in + z_r_in**2/(-f + s_in)) + 1/f)
```

```
>>> gaussian_conj(s_in, z_r_in, f)[1]
z_r_in/(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

```
>>> gaussian_conj(s_in, z_r_in, f)[2]
1/sqrt(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

```
sympy.physics.optics.gaussopt.conjugate_gauss_beams(wavelen, waist_in,
                                                    waist_out, **kwargs)
```

Find the optical setup conjugating the object/image waists.

Parameters **wavelen** : the wavelength of the beam

waist_in and waist_out : the waists to be conjugated

f : the focal distance of the element used in the conjugation

Returns a tuple containing (s_in, s_out, f)

s_in : the distance before the optical element

s_out : the distance after the optical element

f : the focal distance of the optical element

Examples

```
>>> from sympy.physics.optics import conjugate_gauss_beams
>>> from sympy import symbols, factor
>>> l, w_i, w_o, f = symbols('l w_i w_o f')
```

```
>>> conjugate_gauss_beams(l, w_i, w_o, f=f)[0]
f*(-sqrt(w_i**2/w_o**2 - pi**2*w_i**4/(f**2*l**2)) + 1)
```

```
>>> factor(conjugate_gauss_beams(l, w_i, w_o, f=f)[1])
f*w_o**2*(w_i**2/w_o**2 - sqrt(w_i**2/w_o**2 -
                               pi**2*w_i**4/(f**2*l**2)))/w_i**2
```

```
>>> conjugate_gauss_beams(l, w_i, w_o, f=f)[2]
f
```

Medium

Contains

- Medium

class sympy.physics.optics.medium.Medium

This class represents an optical medium. The prime reason to implement this is to facilitate refraction, Fermat's principle, etc.

An optical medium is a material through which electromagnetic waves propagate. The permittivity and permeability of the medium define how electromagnetic waves propagate in it.

Parameters **name: string**

The display name of the Medium.

permittivity: Sympifyable

Electric permittivity of the space.

permeability: Sympifyable

Magnetic permeability of the space.

n: Sympifyable

Index of refraction of the medium.

References

[R428] (page 1873)

Examples

```
>>> from sympy.abc import epsilon, mu
>>> from sympy.physics.optics import Medium
>>> m1 = Medium('m1')
>>> m2 = Medium('m2', epsilon, mu)
>>> m1.intrinsic_impedance
149896229*pi*kilogram*meter**2/(1250000*ampere**2*second**3)
>>> m2.refractive_index
299792458*meter*sqrt(epsilon*mu)/second
```

intrinsic_impedance

Returns intrinsic impedance of the medium.

The intrinsic impedance of a medium is the ratio of the transverse components of the electric and magnetic fields of the electromagnetic wave travelling in the medium. In a region with no electrical conductivity it simplifies to the square root of ratio of magnetic permeability to electric permittivity.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.intrinsic_impedance
149896229*pi*kilogram*meter**2/(1250000*ampere**2*second**3)
```

permeability

Returns magnetic permeability of the medium.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.permeability
pi*kilogram*meter/(2500000*ampere**2*second**2)
```

permittivity

Returns electric permittivity of the medium.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.permittivity
625000*ampere**2*second**4/(22468879468420441*pi*kilogram*meter**3)
```

refractive_index

Returns refractive index of the medium.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.refractive_index
1
```

speed

Returns speed of the electromagnetic wave travelling in the medium.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.speed
299792458*meter/second
```

Utilities

Contains

- `refraction_angle`
- `deviation`
- `brewster_angle`
- `lens_makers_formula`
- `mirror_formula`
- `lens_formula`
- `hyperfocal_distance`
- `transverse_magnification`

`sympy.physics.optics.utils.refraction_angle(incident, medium1, medium2, normal=None, plane=None)`

This function calculates transmitted vector after refraction at planar surface. *medium1* and *medium2* can be *Medium* or any sympifiable object.

If *incident* is an object of *Ray3D*, *normal* also has to be an instance of *Ray3D* in order to get the output as a *Ray3D*. Please note that if plane of separation is not provided and *normal* is an instance of *Ray3D*, *normal* will be assumed to be intersecting incident ray at the plane of separation. This will not be the case when *normal* is a *Matrix* or any other

sequence. If *incident* is an instance of *Ray3D* and *plane* has not been provided and *normal* is not *Ray3D*, output will be a *Matrix*.

Parameters incident : Matrix, Ray3D, or sequence

Incident vector

medium1 : sympy.physics.optics.medium.Medium or sympifiable

Medium 1 or its refractive index

medium2 : sympy.physics.optics.medium.Medium or sympifiable

Medium 2 or its refractive index

normal : Matrix, Ray3D, or sequence

Normal vector

plane : Plane

Plane of separation of the two media.

Examples

```
>>> from sympy.physics.optics import refraction_angle
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> refraction_angle(r1, 1, 1, n)
Matrix([
[ 1],
[ 1],
[-1]])
>>> refraction_angle(r1, 1, 1, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, -1))
```

With different index of refraction of the two media

```
>>> n1, n2 = symbols('n1, n2')
>>> refraction_angle(r1, n1, n2, n)
Matrix([
[                                n1/n2],
[                                n1/n2],
[-sqrt(3)*sqrt(-2*n1**2/(3*n2**2) + 1)]]])
>>> refraction_angle(r1, n1, n2, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(n1/n2, n1/n2, -sqrt(3)*sqrt(-2*n1**2/(3*n2**2) +
↪ 1)))
```

sympy.physics.optics.utils.**deviation**(incident, medium1, medium2, normal=None, plane=None)

This function calculates the angle of deviation of a ray due to refraction at planar surface.

Parameters incident : Matrix, Ray3D, or sequence

Incident vector

medium1 : sympy.physics.optics.medium.Medium or sympifiable

Medium 1 or its refractive index

medium2 : sympy.physics.optics.medium.Medium or sympifiable

Medium 2 or its refractive index

normal : Matrix, Ray3D, or sequence

Normal vector

plane : Plane

Plane of separation of the two media.

Examples

```
>>> from sympy.physics.optics import deviation
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols
>>> n1, n2 = symbols('n1, n2')
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> deviation(r1, 1, 1, n)
0
>>> deviation(r1, n1, n2, plane=P)
-acos(-sqrt(-2*n1**2/(3*n2**2) + 1)) + acos(-sqrt(3)/3)
```

sympy.physics.optics.utils.brewster_angle(medium1, medium2)

This function calculates the Brewster's angle of incidence to Medium 2 from Medium 1 in radians.

Parameters medium 1 : Medium or sympifiable

Refractive index of Medium 1

medium 2 : Medium or sympifiable

Refractive index of Medium 1

Examples

```
>>> from sympy.physics.optics import brewster_angle
>>> brewster_angle(1, 1.33)
0.926093295503462
```

sympy.physics.optics.utils.lens_makers_formula(n_lens, n_surr, r1, r2)

This function calculates focal length of a thin lens. It follows cartesian sign convention.

Parameters n_lens : Medium or sympifiable

Index of refraction of lens.

n_surr : Medium or sympifiable

Index of reflection of surrounding.

r1 : sympifiable

Radius of curvature of first surface.

r2 : sympifiable

Radius of curvature of second surface.

Examples

```
>>> from sympy.physics.optics import lens_makers_formula
>>> lens_makers_formula(1.33, 1, 10, -10)
15.1515151515151
```

`sympy.physics.optics.utils.mirror_formula(focal_length=None, u=None, v=None)`

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

Parameters `focal_length` : sympifiable

Focal length of the mirror.

u : sympifiable

Distance of object from the pole on the principal axis.

v : sympifiable

Distance of the image from the pole on the principal axis.

Examples

```
>>> from sympy.physics.optics import mirror_formula
>>> from sympy.abc import f, u, v
>>> mirror_formula(focal_length=f, u=u)
f*u/(-f + u)
>>> mirror_formula(focal_length=f, v=v)
f*v/(-f + v)
>>> mirror_formula(u=u, v=v)
u*v/(u + v)
```

`sympy.physics.optics.utils.lens_formula(focal_length=None, u=None, v=None)`

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

Parameters `focal_length` : sympifiable

Focal length of the mirror.

u : sympifiable

Distance of object from the optical center on the principal axis.

v : sympifiable

Distance of the image from the optical center on the principal axis.

Examples

```
>>> from sympy.physics.optics import lens_formula
>>> from sympy.abc import f, u, v
>>> lens_formula(focal_length=f, u=u)
f*u/(f + u)
>>> lens_formula(focal_length=f, v=v)
f*v/(f - v)
```

(continues on next page)

(continued from previous page)

```
>>> lens_formula(u=u, v=v)
u*v/(u - v)
```

`sympy.physics.optics.utils.hyperfocal_distance(f, N, c)`

Parameters f: sympifiable

Focal length of a given lens

N: sympifiable

F-number of a given lens

c: sympifiable

Circle of Confusion (CoC) of a given image format

Example

```
>>> from sympy.physics.optics import hyperfocal_distance
>>> from sympy.abc import f, N, c
>>> round(hyperfocal_distance(f = 0.5, N = 8, c = 0.0033), 2)
9.47
```

`sympy.physics.optics.utils.transverse_magnification(si, so)`

Calculates the transverse magnification, which is the ratio of the image size to the object size.

Parameters so: sympifiable

Lens-object distance

si: sympifiable

Lens-image distance

Example

```
>>> from sympy.physics.optics import transverse_magnification
>>> transverse_magnification(30, 15)
-2
```

Waves

This module has all the classes and functions related to waves in optics.

Contains

- TWave

class `sympy.physics.optics.waves.TWave(amplitude, frequency=None, phase=0, time_period=None, n=n)`

This is a simple transverse sine wave travelling in a one-dimensional space. Basic properties are required at the time of creation of the object, but they can be changed later with respective methods provided.

It is represented as $A \times \cos(k \times x - \omega \times t + \phi)$, where A is the amplitude, ω is the angular velocity, k is the wavenumber (spatial frequency), x is a spatial variable to represent the position on the dimension on which the wave propagates, and ϕ is the phase angle of the wave.

Raises ValueError : When neither frequency nor time period is provided
or they are not consistent.

TypeError : When anything other than TWave objects is added.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A1, phi1, A2, phi2, f = symbols('A1, phi1, A2, phi2, f')
>>> w1 = TWave(A1, f, phi1)
>>> w2 = TWave(A2, f, phi2)
>>> w3 = w1 + w2 # Superposition of two waves
>>> w3
TWave(sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2), f,
      atan2(A1*cos(phi1) + A2*cos(phi2), A1*sin(phi1) + A2*sin(phi2)))
>>> w3.amplitude
sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2)
>>> w3.phase
atan2(A1*cos(phi1) + A2*cos(phi2), A1*sin(phi1) + A2*sin(phi2))
>>> w3.speed
299792458*meter/(second*n)
>>> w3.angular_velocity
2*pi*f
```

Arguments

amplitude [Sympifyable] Amplitude of the wave.

frequency [Sympifyable] Frequency of the wave.

phase [Sympifyable] Phase angle of the wave.

time_period [Sympifyable] Time period of the wave.

n [Sympifyable] Refractive index of the medium.

amplitude

Returns the amplitude of the wave.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.amplitude
A
```

angular_velocity

Returns the angular velocity of the wave, in radians per second.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.angular_velocity
2*pi*f
```

frequency

Returns the frequency of the wave, in cycles per second.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.frequency
f
```

phase

Returns the phase angle of the wave, in radians.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.phase
phi
```

speed

Returns the propagation speed of the wave, in meters per second. It is dependent on the propagation medium.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.speed
299792458*meter/(second*n)
```

time_period

Returns the temporal period of the wave, in seconds per cycle.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.time_period
1/f
```

wavelength

Returns the wavelength (spatial period) of the wave, in meters per cycle. It depends on the medium of the wave.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavelength
299792458*meter/(second*f*n)
```

wavenumber

Returns the wavenumber of the wave, in radians per meter.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavenumber
pi*second*f*n/(149896229*meter)
```

Continuum Mechanics

Abstract

Contains docstrings for methods in continuum mechanics module.

Beam

Beam (Docstrings)

Beam

This module can be used to solve 2D beam bending problems with singularity functions in mechanics.


```
class sympy.physics.continuum_mechanics.beam.Beam(length, elastic_modulus,
                                                    second_moment, variable=x)
```

A Beam is a structural element that is capable of withstanding load primarily by resisting against bending. Beams are characterized by their cross sectional profile(Second moment of area), their length and their material.

Note: While solving a beam bending problem, a user should choose its own sign convention and should stick to it. The results will automatically follow the chosen sign convention.

Examples

There is a beam of length 4 meters. A constant distributed load of 6 N/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. The deflection of the beam at the end is restricted.

Using the sign convention of downwards forces being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, Piecewise
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(4, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(6, 2, 0)
>>> b.apply_load(R2, 4, -1)
>>> b.bc_deflection = [(0, 0), (4, 0)]
>>> b.boundary_conditions
{'deflection': [(0, 0), (4, 0)], 'slope': []}
>>> b.load
R1*SingularityFunction(x, 0, -1) + R2*SingularityFunction(x, 4, -1) +
↳ 6*SingularityFunction(x, 2, 0)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.load
-3*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 2, 0) -
↳ 9*SingularityFunction(x, 4, -1)
>>> b.shear_force()
-3*SingularityFunction(x, 0, 0) + 6*SingularityFunction(x, 2, 1) -
↳ 9*SingularityFunction(x, 4, 0)
>>> b.bending_moment()
-3*SingularityFunction(x, 0, 1) + 3*SingularityFunction(x, 2, 2) -
↳ 9*SingularityFunction(x, 4, 1)
>>> b.slope()
(-3*SingularityFunction(x, 0, 2)/2 + SingularityFunction(x, 2, 3) -
↳ 9*SingularityFunction(x, 4, 2)/2 + 7)/(E*I)
>>> b.deflection()
(7*x - SingularityFunction(x, 0, 3)/2 + SingularityFunction(x, 2, 4)/4 -
↳ 3*SingularityFunction(x, 4, 3)/2)/(E*I)
>>> b.deflection().rewrite(Piecewise)
(7*x - Piecewise((x**3, x > 0), (0, True))/2
- 3*Piecewise(((x - 4)**3, x - 4 > 0), (0, True))/2
+ Piecewise(((x - 2)**4, x - 2 > 0), (0, True))/4)/(E*I)
```

applied_loads

Returns a list of all loads applied on the beam object. Each load in the list is a tuple of form (value, start, order, end).

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point. Another pointload of magnitude 5 N is applied at same position.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(5, 2, -1)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 9*SingularityFunction(x, 2, -1)
>>> b.applied_loads
[(-3, 0, -2, None), (4, 2, -1, None), (5, 2, -1, None)]
```

apply_load(value, start, order, end=None)

This method adds up the loads given to a particular beam object.

Parameters value : Sympifyable

The magnitude of an applied load.

start : Sympifyable

The starting point of the applied load. For point moments and point forces this is the location of application.

order : Integer

The order of the applied load. - For moments, order=-2 - For point loads, order=-1 - For constant distributed load, order=0 - For ramp loads, order=1 - For parabolic ramp loads, order=2 - ... so on.

end : Sympifyable, optional

An optional argument that can be used if the load has an end point within the length of the beam.

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2 meters to 3 meters away from the starting point of the beam.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
```

(continues on next page)

(continued from previous page)

```

>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end = 3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
↳ 2*SingularityFunction(x, 2, 2)
  + 2*SingularityFunction(x, 3, 0) + 2*SingularityFunction(x, 3, 2)

```

apply_support(loc, type='fixed')

This method applies support to a particular beam object.

Parameters **loc** : Sympifyable

Location of point at which support is applied.

type : String

Determines type of Beam support applied. To apply support structure with - zero degree of freedom, type = "fixed" - one degree of freedom, type = "pin" - two degrees of freedom, type = "roller"

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(30, E, I)
>>> b.apply_support(10, 'roller')
>>> b.apply_support(30, 'roller')
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(120, 30, -2)
>>> R_10, R_30 = symbols('R_10, R_30')
>>> b.solve_for_reaction_loads(R_10, R_30)
>>> b.load
-8*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 10, -1)
+ 120*SingularityFunction(x, 30, -2) + 2*SingularityFunction(x, 30, -1)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2)
+ 120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30, 2) +
↳ 4000/3)/(E*I)

```

bending_moment()

Returns a Singularity Function expression which represents the bending moment curve of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.bending_moment()
-8*SingularityFunction(x, 0, 1) + 6*SingularityFunction(x, 10, 1) +
120*SingularityFunction(x, 30, 0) + 2*SingularityFunction(x, 30, 1)
```

boundary_conditions

Returns a dictionary of boundary conditions applied on the beam. The dictionary has three keywords namely moment, slope and deflection. The value of each keyword is a list of tuple, where each tuple contains location and value of a boundary condition in the format (location, value).

Examples

There is a beam of length 4 meters. The bending moment at 0 should be 4 and at 4 it should be 0. The slope of the beam should be 1 at 0. The deflection should be 2 at 0.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.bc_deflection = [(0, 2)]
>>> b.bc_slope = [(0, 1)]
>>> b.boundary_conditions
{'deflection': [(0, 2)], 'slope': [(0, 1)]}
```

Here the deflection of the beam should be 2 at 0. Similarly, the slope of the beam should be 1 at 0.

deflection()

Returns a Singularity Function expression which represents the elastic curve or deflection of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.deflection()
(4000*x/3 - 4*SingularityFunction(x, 0, 3)/3 + SingularityFunction(x, 10, 3)
 + 60*SingularityFunction(x, 30, 2) + SingularityFunction(x, 30, 3)/3 -
 ↪ 12000)/(E*I)
```

elastic_modulus

Young's Modulus of the Beam.

join(*beam*, *via*='fixed')

This method joins two beams to make a new composite beam system. Passed Beam class instance is attached to the right end of calling object.

Parameters *beam* : Beam class object

The Beam object which would be connected to the right of calling object.

via : String

States the way two Beam object would get connected - For axially fixed Beams, *via*="fixed" - For Beams connected via hinge, *via*="hinge"

Examples

There is a cantilever beam of length 4 meters. For first 2 meters its moment of inertia is $1.5*I$ and I for the other end. A pointload of magnitude 4 N is applied from the top at its free end.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b1 = Beam(2, E, 1.5*I)
>>> b2 = Beam(2, E, I)
>>> b = b1.join(b2, "fixed")
>>> b.apply_load(20, 4, -1)
```

(continues on next page)

(continued from previous page)

```

>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 0, -2)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.load
80*SingularityFunction(x, 0, -2) - 20*SingularityFunction(x, 0, -1) +
↳ 20*SingularityFunction(x, 4, -1)
>>> b.slope()
Piecewise((0.6666666666666667*(80*SingularityFunction(x, 0, 1) -
↳ 10*SingularityFunction(x, 0, 2) +
10*SingularityFunction(x, 4, 2))/(E*I), x <= 2), ((80*SingularityFunction(x,
↳ 0, 1) -
10*SingularityFunction(x, 0, 2) + 10*SingularityFunction(x, 4, 2))/I - 120/I)/
↳ E + 80.0/(E*I), x <= 4))

```

length

Length of the Beam.

load

Returns a Singularity Function expression which represents the load distribution curve of the Beam object.

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 3 meters away from the starting point of the beam.

```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 3, 2)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
↳ 2*SingularityFunction(x, 3, 2)

```

max_bmoment()

Returns maximum Shear force and its coordinate in the Beam object.

max_deflection()

Returns point of max deflection and its corresponding deflection value in a Beam object.

max_shear_force()

Returns maximum Shear force and its coordinate in the Beam object.

point_cflexure()

Returns a Set of point(s) with zero bending moment and where bending moment curve of the beam object changes its sign from negative to positive or vice versa.

Examples

There is a 10 meter long overhanging beam. There are two simple supports below the beam. One at the start and another one at a distance of 6 meters from the start. Point loads of magnitude 10KN and 20KN are applied at 2 meters and 4 meters from start respectively. A Uniformly distributed load of magnitude of 3KN/m is also applied on top starting from 6 meters away from starting point till end. Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(10, E, I)
>>> b.apply_load(-4, 0, -1)
>>> b.apply_load(-46, 6, -1)
>>> b.apply_load(10, 2, -1)
>>> b.apply_load(20, 4, -1)
>>> b.apply_load(3, 6, 0)
>>> b.point_cflexure()
[10/3]
```

reaction_loads

Returns the reaction forces in a dictionary.

remove_load(value, start, order, end=None)

This method removes a particular load present on the beam object. Returns a ValueError if the load passed as an argument is not present on the beam.

Parameters value : Sympifyable

The magnitude of an applied load.

start : Sympifyable

The starting point of the applied load. For point moments and point forces this is the location of application.

order : Integer

The order of the applied load. - For moments, order= -2 - For point loads, order=-1 - For constant distributed load, order=0 - For ramp loads, order=1 - For parabolic ramp loads, order=2 - ... so on.

end : Sympifyable, optional

An optional argument that can be used if the load has an end point within the length of the beam.

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A pointload of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2 meters to 3 meters away from the starting point of the beam.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
```

(continues on next page)

(continued from previous page)

```

>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end = 3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
↳ 2*SingularityFunction(x, 2, 2)
  + 2*SingularityFunction(x, 3, 0) + 2*SingularityFunction(x, 3, 2)
>>> b.remove_load(-2, 2, 2, end = 3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1)

```

second_moment

Second moment of area of the Beam.

shear_force()

Returns a Singularity Function expression which represents the shear force curve of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```

>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.shear_force()
-8*SingularityFunction(x, 0, 0) + 6*SingularityFunction(x, 10, 0) +
↳ 120*SingularityFunction(x, 30, -1) + 2*SingularityFunction(x, 30, 0)

```

slope()

Returns a Singularity Function expression which represents the slope the elastic curve of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports

below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2)
 + 120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30, 2) +
 4000/3)/(E*I)
```

solve_for_reaction_loads(*reactions)

Solves for the reaction forces.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, linsolve, limit
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1) # Reaction force at x = 10
>>> b.apply_load(R2, 30, -1) # Reaction force at x = 30
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.load
R1*SingularityFunction(x, 10, -1) + R2*SingularityFunction(x, 30, -1)
- 8*SingularityFunction(x, 0, -1) + 120*SingularityFunction(x, 30, -2)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: 6, R2: 2}
>>> b.load
-8*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 10, -1)
+ 120*SingularityFunction(x, 30, -2) + 2*SingularityFunction(x, 30, -1)
```

variable

A symbol that can be used as a variable along the length of the beam while representing load distribution, shear force curve, bending moment, slope curve and the deflection curve. By default, it is set to `Symbol('x')`, but this property is mutable.

Examples

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> x, y, z = symbols('x, y, z')
>>> b = Beam(4, E, I)
>>> b.variable
x
>>> b.variable = y
>>> b.variable
y
>>> b = Beam(4, E, I, z)
>>> b.variable
z
```

Solving Beam Bending Problems using Singularity Functions

To make this document easier to read, we are going to enable pretty printing.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

Beam

A Beam is a structural element that is capable of withstanding load primarily by resisting against bending. Beams are characterized by their second moment of area, their length and their elastic modulus.

In SymPy, we can construct a 2D beam objects with the following properties :

- Length
- Elastic Modulus
- Second Moment of Area
- Variable : A symbol that can be used as a variable along the length. By default, this is set to `Symbol(x)`.
- **Boundary Conditions**
 - `bc_slope` : Boundary conditions for slope.
 - `bc_deflection` : Boundary conditions for deflection.
- Load Distribution

We have following methods under the beam class:

- `apply_load`
- `solve_for_reaction_loads`
- `shear_force`
- `bending_moment`
- `slope`

- deflection

Examples

Let us solve some beam bending problems using this module :

Example 1

A beam of length 9 meters is having a fixed support at the start. A distributed constant load of 8 kN/m is applied downward from the starting point till 5 meters away from the start. A clockwise moment of 50 kN-m is applied at 5 meters away from the start of the beam. A downward point load of 12 kN is applied at the end.

Note: Since a user is free to choose its own sign convention we are considering the upward forces and clockwise bending moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, M1 = symbols('R1, M1')
>>> b = Beam(9, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(-8, 0, 0, end=5)
>>> b.apply_load(50, 5, -2)
>>> b.apply_load(-12, 9, -1)
>>> b.bc_slope.append((0, 0))
>>> b.bc_deflection.append((0, 0))
>>> b.solve_for_reaction_loads(R1, M1)
>>> b.reaction_loads
{M1: -258, R1: 52}
>>> b.load
      -2      -1      0      -2      0      -1
- 258·<x> + 52·<x> - 8·<x> + 50·<x - 5> + 8·<x - 5> - 12·<x - 9>
>>> b.shear_force()
      -1      0      1      -1      1      0
- 258·<x> + 52·<x> - 8·<x> + 50·<x - 5> + 8·<x - 5> - 12·<x - 9>
>>> b.bending_moment()
      0      1      2      0      2      1
- 258·<x> + 52·<x> - 4·<x> + 50·<x - 5> + 4·<x - 5> - 12·<x - 9>
>>> b.slope()
      1      2      3      1      3      2
- 258·<x> + 26·<x> -  $\frac{4 \cdot \langle x \rangle^3}{3}$  + 50·<x - 5> +  $\frac{4 \cdot \langle x - 5 \rangle^3}{3}$  - 6·<x - 9>
      _____
                        E·I
>>> b.deflection()
      2      3      4      2      4      3
- 129·<x> +  $\frac{26 \cdot \langle x \rangle^3}{3}$  -  $\frac{\langle x \rangle^4}{3}$  + 25·<x - 5> +  $\frac{\langle x - 5 \rangle^4}{3}$  - 2·<x - 9>
```

(continues on next page)

(continued from previous page)

$$E \cdot I$$

Example 2

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Note: Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection.append((10, 0))
>>> b.bc_deflection.append((30, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: 6, R2: 2}
>>> b.load
      -1      -1      -2      -1
      8·<x> + 6·<x - 10> + 120·<x - 30> + 2·<x - 30>
>>> b.shear_force()
      0      0      -1      0
      8·<x> + 6·<x - 10> + 120·<x - 30> + 2·<x - 30>
>>> b.bending_moment()
      1      1      0      1
      8·<x> + 6·<x - 10> + 120·<x - 30> + 2·<x - 30>
>>> b.slope()
      2      2      1      2      4000
      4·<x> + 3·<x - 10> + 120·<x - 30> + <x - 30> +  $\frac{4000}{3}$ 
      -----
                        E·I
>>> b.deflection()
      3      3      2      3
      4000·x -  $\frac{4·<x>^3}{3}$  + <x - 10>3 + 60·<x - 30>2 +  $\frac{<x - 30>^3}{3}$  - 12000
      -----
                        E·I
```

Example 3

A beam of length 6 meters is having a roller support at the start and a hinged support at the end. A clockwise moment of 1.5 kN-m is applied at the mid of the beam. A constant distributed load of 3 kN/m and a ramp load of 1 kN/m is applied from the mid til the end of the beam.

Note: Using the sign convention of upward forces and clockwise moment being positive.

```
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(6, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(1.5, 3, -2)
>>> b.apply_load(-3, 3, 0)
>>> b.apply_load(-1, 3, 1)
>>> b.apply_load(R2, 6, -1)
>>> b.bc_deflection.append((0, 0))
>>> b.bc_deflection.append((6, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: 2.75, R2: 10.75}
>>> b.load

$$2.75 \cdot \langle x \rangle^{-1} + 1.5 \cdot \langle x - 3 \rangle^{-2} - 3 \cdot \langle x - 3 \rangle^0 - \langle x - 3 \rangle^1 + 10.75 \cdot \langle x - 6 \rangle^{-1}$$

>>> b.shear_force()

$$2.75 \cdot \langle x \rangle^0 + 1.5 \cdot \langle x - 3 \rangle^{-1} - 3 \cdot \langle x - 3 \rangle^1 - \frac{\langle x - 3 \rangle^2}{2} + 10.75 \cdot \langle x - 6 \rangle^0$$

>>> b.bending_moment()

$$2.75 \cdot \langle x \rangle^1 + 1.5 \cdot \langle x - 3 \rangle^0 - \frac{3 \cdot \langle x - 3 \rangle^2}{2} - \frac{\langle x - 3 \rangle^3}{6} + 10.75 \cdot \langle x - 6 \rangle^1$$

>>> b.slope()

$$\frac{1.375 \cdot \langle x \rangle^2 + 1.5 \cdot \langle x - 3 \rangle^1 - \frac{\langle x - 3 \rangle^3}{2} - \frac{\langle x - 3 \rangle^4}{24} + 5.375 \cdot \langle x - 6 \rangle^2 - 15.6}{E \cdot I}$$

>>> b.deflection()

$$\frac{-15.6 \cdot x^3 + 0.458333333333333 \cdot \langle x \rangle^3 + 0.75 \cdot \langle x - 3 \rangle^2 - \frac{\langle x - 3 \rangle^4}{8} - \frac{\langle x - 3 \rangle^5}{120} + 1.79166666666667 \cdot \langle x - 6 \rangle^3}{E \cdot I}$$

```

5.40 Category Theory Module

5.40.1 Introduction

The category theory module for SymPy will allow manipulating diagrams within a single category, including drawing them in TikZ and deciding whether they are commutative or not.

The general reference work this module tries to follow is

[JoyOfCats] J. Adamek, H. Herrlich. G. E. Strecker: Abstract and Concrete Categories. The Joy of Cats.

The latest version of this book should be available for free download from

katmat.math.uni-bremen.de/acc/acc.pdf

The module is still in its pre-embryonic stage.

5.40.2 Base Class Reference

This section lists the classes which implement some of the basic notions in category theory: objects, morphisms, categories, and diagrams.

class `sympy.categories.Object`

The base class for any kind of object in an abstract category.

While technically any instance of `Basic` will do, this class is the recommended way to create abstract objects in abstract categories.

class `sympy.categories.Morphism`

The base class for any morphism in an abstract category.

In abstract categories, a morphism is an arrow between two category objects. The object where the arrow starts is called the domain, while the object where the arrow ends is called the codomain.

Two morphisms between the same pair of objects are considered to be the same morphisms. To distinguish between morphisms between the same objects use `NamedMorphism` (page 1737).

It is prohibited to instantiate this class. Use one of the derived classes instead.

See also:

`IdentityMorphism` (page 1740), `NamedMorphism` (page 1737), `CompositeMorphism` (page 1738)

codomain

Returns the codomain of the morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.codomain
Object("B")
```

compose(other)

Composes self with the supplied morphism.

The order of elements in the composition is the usual order, i.e., to construct $g \circ f$ use `g.compose(f)`.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> (g * f).domain
Object("A")
>>> (g * f).codomain
Object("C")
```

domain

Returns the domain of the morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.domain
Object("A")
```

class `sympy.categories.NamedMorphism`

Represents a morphism which has a name.

Names are used to distinguish between morphisms which have the same domain and codomain: two named morphisms are equal if they have the same domains, codomains, and names.

See also:

[Morphism](#) (page 1736)

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f
NamedMorphism(Object("A"), Object("B"), "f")
```

(continues on next page)

(continued from previous page)

```
>>> f.name
'f'
```

name

Returns the name of the morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.name
'f'
```

class sympy.categories.CompositeMorphism

Represents a morphism which is a composition of other morphisms.

Two composite morphisms are equal if the morphisms they were obtained from (components) are the same and were listed in the same order.

The arguments to the constructor for this class should be listed in diagram order: to obtain the composition $g \circ f$ from the instances of `Morphism` (page 1736) `g` and `f` use `CompositeMorphism(f, g)`.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, CompositeMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> CompositeMorphism(f, g) == g * f
True
```

codomain

Returns the codomain of this composite morphism.

The codomain of the composite morphism is the codomain of its last component.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
```

(continues on next page)

(continued from previous page)

```
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).codomain
Object("C")
```

components

Returns the components of this composite morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).components
(NamedMorphism(Object("A"), Object("B"), "f"),
 NamedMorphism(Object("B"), Object("C"), "g"))
```

domain

Returns the domain of this composite morphism.

The domain of the composite morphism is the domain of its first component.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).domain
Object("A")
```

flatten(new_name)

Forgets the composite structure of this morphism.

If new_name is not empty, returns a [NamedMorphism](#) (page 1737) with the supplied name, otherwise returns a [Morphism](#) (page 1736). In both cases the domain of the new morphism is the domain of this composite morphism and the codomain of the new morphism is the codomain of this composite morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
```

(continues on next page)

(continued from previous page)

```
>>> (g * f).flatten("h")
NamedMorphism(Object("A"), Object("C"), "h")
```

class sympy.categories.IdentityMorphism

Represents an identity morphism.

An identity morphism is a morphism with equal domain and codomain, which acts as an identity with respect to composition.

See also:

[Morphism](#) (page 1736)

Examples

```
>>> from sympy.categories import Object, NamedMorphism, IdentityMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> f * id_A == f
True
>>> id_B * f == f
True
```

class sympy.categories.Category

An (abstract) category.

A category [JoyOfCats] is a quadruple $K = (O, \text{hom}, id, \circ)$ consisting of

- a (set-theoretical) class O , whose members are called K -objects,
- for each pair (A, B) of K -objects, a set $\text{hom}(A, B)$ whose members are called K -morphisms from A to B ,
- for a each K -object A , a morphism $id : A \rightarrow A$, called the K -identity of A ,
- a composition law $\circ$ associating with every K -morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$ a K -morphism $g \circ f : A \rightarrow C$, called the composite of f and g .

Composition is associative, K -identities are identities with respect to composition, and the sets $\text{hom}(A, B)$ are pairwise disjoint.

This class knows nothing about its objects and morphisms. Concrete cases of (abstract) categories should be implemented as classes derived from this one.

Certain instances of [Diagram](#) (page 1741) can be asserted to be commutative in a [Category](#) (page 1740) by supplying the argument `commutative_diagrams` in the constructor.

See also:

[Diagram](#) (page 1741)

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

commutative_diagrams

Returns the FiniteSet of diagrams which are known to be commutative in this category.

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

name

Returns the name of this category.

Examples

```
>>> from sympy.categories import Category
>>> K = Category("K")
>>> K.name
'K'
```

objects

Returns the class of objects of this category.

Examples

```
>>> from sympy.categories import Object, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> K = Category("K", FiniteSet(A, B))
>>> K.objects
Class({Object("A"), Object("B")})
```

class sympy.categories.Diagram

Represents a diagram in a certain category.

Informally, a diagram is a collection of objects of a category and certain morphisms between them. A diagram is still a monoid with respect to morphism composition; i.e., identity morphisms, as well as all composites of morphisms included in the diagram belong to the diagram. For a more formal approach to this notion see [Pare1970].

The components of composite morphisms are also added to the diagram. No properties are assigned to such morphisms by default.

A commutative diagram is often accompanied by a statement of the following kind: “if such morphisms with such properties exist, then such morphisms which such properties exist and the diagram is commutative”. To represent this, an instance of `Diagram` (page 1741) includes a collection of morphisms which are the premises and another collection of conclusions. `premises` and `conclusions` associate morphisms belonging to the corresponding categories with the `FiniteSet`’s of their properties.

The set of properties of a composite morphism is the intersection of the sets of properties of its components. The domain and codomain of a conclusion morphism should be among the domains and codomains of the morphisms listed as the premises of a diagram.

No checks are carried out of whether the supplied object and morphisms do belong to one and the same category.

References

[Pare1970] B. Pareigis: Categories and functors. Academic Press, 1970.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet, pprint, default_sort_key
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> premises_keys = sorted(d.premises.keys(), key=default_sort_key)
>>> pprint(premises_keys, use_unicode=False)
[g*f:A-->C, id:A-->A, id:B-->B, id:C-->C, f:A-->B, g:B-->C]
>>> pprint(d.premises, use_unicode=False)
{g*f:A-->C: EmptySet(), id:A-->A: EmptySet(), id:B-->B: EmptySet(), id:C-->C:
EmptySet(), f:A-->B: EmptySet(), g:B-->C: EmptySet()}
>>> d = Diagram([f, g], {g * f: "unique"})
>>> pprint(d.conclusions)
{g*f:A-->C: {unique}}
```

conclusions

Returns the conclusions of this diagram.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
```

(continues on next page)

(continued from previous page)

```

>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> IdentityMorphism(A) in d.premises.keys()
True
>>> g * f in d.premises.keys()
True
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d.conclusions[g * f] == FiniteSet("unique")
True

```

hom(A, B)

Returns a 2-tuple of sets of morphisms between objects A and B: one set of morphisms listed as premises, and the other set of morphisms listed as conclusions.

See also:

[Object](#) (page 1736), [Morphism](#) (page 1736)

Examples

```

>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> print(pretty(d.hom(A, C), use_unicode=False))
({g*f:A-->C}, {g*f:A-->C})

```

is_subdiagram(diagram)

Checks whether diagram is a subdiagram of self. Diagram D' is a subdiagram of D if all premises (conclusions) of D' are contained in the premises (conclusions) of D . The morphisms contained both in D' and D should have the same properties for D' to be a subdiagram of D .

Examples

```

>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d1 = Diagram([f])
>>> d.is_subdiagram(d1)

```

(continues on next page)

(continued from previous page)

```
True
>>> d1.is_subdiagram(d)
False
```

objects

Returns the FiniteSet of objects that appear in this diagram.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> d.objects
{Object("A"), Object("B"), Object("C")}
```

premises

Returns the premises of this diagram.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> d = Diagram([f])
>>> print(pretty(d.premises, use_unicode=False))
{id:A-->A: EmptySet(), id:B-->B: EmptySet(), f:A-->B: EmptySet()}
```

subdiagram_from_objects(objects)

If objects is a subset of the objects of self, returns a diagram which has as premises all those premises of self which have a domains and codomains in objects, likewise for conclusions. Properties are preserved.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
```

(continues on next page)

(continued from previous page)

```
>>> d = Diagram([f, g], {f: "unique", g*f: "veryunique"})
>>> d1 = d.subdiagram_from_objects(FiniteSet(A, B))
>>> d1 == Diagram([f], {f: "unique"})
True
```

5.40.3 Diagram Drawing

This section lists the classes which allow automatic drawing of diagrams.

class sympy.categories.diagram_drawing.**DiagramGrid**(diagram, groups=None, **hints)

Constructs and holds the fitting of the diagram into a grid.

The mission of this class is to analyse the structure of the supplied diagram and to place its objects on a grid such that, when the objects and the morphisms are actually drawn, the diagram would be “readable”, in the sense that there will not be many intersections of morphisms. This class does not perform any actual drawing. It does strive nevertheless to offer sufficient metadata to draw a diagram.

Consider the following simple diagram.

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> from sympy import pprint
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
```

The simplest way to have a diagram laid out is the following:

```
>>> grid = DiagramGrid(diagram)
>>> (grid.width, grid.height)
(2, 2)
>>> pprint(grid)
A  B
   C
```

Sometimes one sees the diagram as consisting of logical groups. One can advise `DiagramGrid` as to such groups by employing the `groups` keyword argument.

Consider the following diagram:

```
>>> D = Object("D")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
```

Lay it out with generic layout:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A  B  D
   C
```

Now, we can group the objects *A* and *D* to have them near one another:

```
>>> grid = DiagramGrid(diagram, groups=[[A, D], B, C])
>>> pprint(grid)
B      C
A  D
```

Note how the positioning of the other objects changes.

Further indications can be supplied to the constructor of [DiagramGrid](#) (page 1745) using keyword arguments. The currently supported hints are explained in the following paragraphs.

[DiagramGrid](#) (page 1745) does not automatically guess which layout would suit the supplied diagram better. Consider, for example, the following linear diagram:

```
>>> E = Object("E")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(C, D, "h")
>>> i = NamedMorphism(D, E, "i")
>>> diagram = Diagram([f, g, h, i])
```

When laid out with the generic layout, it does not get to look linear:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A  B
   C  D
   E
```

To get it laid out in a line, use `layout="sequential"`:

```
>>> grid = DiagramGrid(diagram, layout="sequential")
>>> pprint(grid)
A  B  C  D  E
```

One may sometimes need to transpose the resulting layout. While this can always be done by hand, [DiagramGrid](#) (page 1745) provides a hint for that purpose:

```
>>> grid = DiagramGrid(diagram, layout="sequential", transpose=True)
>>> pprint(grid)
A
B
C
D
```

(continues on next page)

(continued from previous page)

E

Separate hints can also be provided for each group. For an example, refer to `tests/test_drawing.py`, and see the different ways in which the five lemma [FiveLemma] can be laid out.

See also:

Diagram

References[FiveLemma] http://en.wikipedia.org/wiki/Five_lemma**height**

Returns the number of rows in this diagram layout.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.height
2
```

morphisms

Returns those morphisms (and their properties) which are sufficiently meaningful to be drawn.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.morphisms
{NamedMorphism(Object("A"), Object("B"), "f"): EmptySet(),
 NamedMorphism(Object("B"), Object("C"), "g"): EmptySet()}
```

width

Returns the number of columns in this diagram layout.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.width
2
```

```
class sympy.categories.diagram_drawing.ArrowStringDescription(unit, curving,
                                                              curving_amount,
                                                              looping_start,
                                                              looping_end,
                                                              horizontal_direction,
                                                              vertical_direction,
                                                              label_position,
                                                              label)
```

Stores the information necessary for producing an Xy-pic description of an arrow.

The principal goal of this class is to abstract away the string representation of an arrow and to also provide the functionality to produce the actual Xy-pic string.

`unit` sets the unit which will be used to specify the amount of curving and other distances. `horizontal_direction` should be a string of "r" or "l" specifying the horizontal offset of the target cell of the arrow relatively to the current one. `vertical_direction` should specify the vertical offset using a series of either "d" or "u". `label_position` should be either "^", "_", or "|" to specify that the label should be positioned above the arrow, below the arrow or just over the arrow, in a break. Note that the notions "above" and "below" are relative to arrow direction. `label` stores the morphism label.

This works as follows (disregard the yet unexplained arguments):

```
>>> from sympy.categories.diagram_drawing import ArrowStringDescription
>>> astr = ArrowStringDescription(
... unit="mm", curving=None, curving_amount=None,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar[dr]_{f}
```

`curving` should be one of "^", "_" to specify in which direction the arrow is going to curve. `curving_amount` is a number describing how many unit's the morphism is going to curve:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
```

(continues on next page)

(continued from previous page)

```
>>> print(str(astr))
\ar@/^12mm/[dr]_{f}
```

looping_start and looping_end are currently only used for loop morphisms, those which have the same domain and codomain. These two attributes should store a valid Xy-pic direction and specify, correspondingly, the direction the arrow gets out into and the direction the arrow gets back from:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving=None, curving_amount=None,
... looping_start="u", looping_end="l", horizontal_direction="",
... vertical_direction="", label_position="_", label="f")
>>> print(str(astr))
\ar@(u,l)[l]_{f}
```

label_displacement controls how far the arrow label is from the ends of the arrow. For example, to position the arrow label near the arrow head, use ">":

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.label_displacement = ">"
>>> print(str(astr))
\ar@/^12mm/[dr]_{>f}
```

Finally, arrow_style is used to specify the arrow style. To get a dashed arrow, for example, use "{-->}" as arrow style:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.arrow_style = "{-->}"
>>> print(str(astr))
\ar@/^12mm/@{-->}[dr]_{f}
```

See also:

[XypicDiagramDrawer](#) (page 1749)

Notes

Instances of [ArrowStringDescription](#) (page 1748) will be constructed by [XypicDiagramDrawer](#) (page 1749) and provided for further use in formatters. The user is not expected to construct instances of [ArrowStringDescription](#) (page 1748) themselves.

To be able to properly utilise this class, the reader is encouraged to checkout the Xy-pic user guide, available at [\[Xypic\]](#).

References

[Xypic] <http://xy-pic.sourceforge.net/>

class sympy.categories.diagram_drawing.XypicDiagramDrawer

Given a [Diagram](#) and the corresponding [DiagramGrid](#) (page 1745), produces the Xy-pic representation of the diagram.

The most important method in this class is `draw`. Consider the following triangle diagram:

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a [DiagramGrid](#) (page 1745):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[d]_{g \circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

For further details see the docstring of this method.

To control the appearance of the arrows, formatters are used. The dictionary `arrow_formatters` maps morphisms to formatter functions. A formatter accepts an [ArrowStringDescription](#) (page 1748) and is allowed to modify any of the arrow properties exposed thereby. For example, to have all morphisms with the property `unique` appear as dashed arrows, and to have their names prepended with $\exists!$, the following should be done:

```
>>> def formatter(astr):
...     astr.label = "\exists !" + astr.label
...     astr.arrow_style = "{-->}"
>>> drawer.arrow_formatters["unique"] = formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar@{-->}[d]_{\exists !g \circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

To modify the appearance of all arrows in the diagram, set `default_arrow_formatter`. For example, to place all morphism labels a little bit farther from the arrow head so that they look more centred, do as follows:

```
>>> def default_formatter(astr):
...     astr.label_displacement = "(0.45)"
>>> drawer.default_arrow_formatter = default_formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar@{-->}[d]_{(0.45)\exists !g \circ f} \ar[r]^{(0.45)f} & B \ar[ld]^{(0.45)g} \\
```

(continues on next page)

(continued from previous page)

```
C &
}
```

In some diagrams some morphisms are drawn as curved arrows. Consider the following diagram:

```
>>> D = Object("D")
>>> E = Object("E")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
>>> grid = DiagramGrid(diagram)
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_3mm/[ll]_{h} \\
& C &
}
```

To control how far the morphisms are curved by default, one can use the `unit` and `default_curving_amount` attributes:

```
>>> drawer.unit = "cm"
>>> drawer.default_curving_amount = 1
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_1cm/[ll]_{h} \\
& C &
}
```

In some diagrams, there are multiple curved morphisms between the same two objects. To control by how much the curving changes between two such successive morphisms, use `default_curving_step`:

```
>>> drawer.default_curving_step = 1
>>> h1 = NamedMorphism(A, D, "h1")
>>> diagram = Diagram([f, g, h, k, h1])
>>> grid = DiagramGrid(diagram)
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} \ar@/^1cm/[rr]^{h_1} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_2cm/[ll]_{h} \\
& C &
}
```

The default value of `default_curving_step` is 4 units.

See also:

[draw](#) (page 1751), [ArrowStringDescription](#) (page 1748)

draw(diagram, grid, masked=None, diagram_format="")

Returns the Xy-pic representation of diagram laid out in grid.

Consider the following simple triangle diagram.

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
```

(continues on next page)

(continued from previous page)

```
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a [DiagramGrid](#) (page 1745):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

The argument `masked` can be used to skip morphisms in the presentation of the diagram:

```
>>> print(drawer.draw(diagram, grid, masked=[g * f]))
\xymatrix{
A \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

Finally, the `diagram_format` argument can be used to specify the format string of the diagram. For example, to increase the spacing by 1 cm, proceeding as follows:

```
>>> print(drawer.draw(diagram, grid, diagram_format="@+1cm"))
\xymatrix@+1cm{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[ld]^{g} \\
C & \\
}
```

```
sympy.categories.diagram_drawing.xypic_draw_diagram(diagram, masked=None,
                                                    diagram_format="",
                                                    groups=None, **hints)
```

Provides a shortcut combining [DiagramGrid](#) (page 1745) and [XypicDiagramDrawer](#) (page 1749). Returns an Xy-pic presentation of diagram. The argument `masked` is a list of morphisms which will be not be drawn. The argument `diagram_format` is the format string inserted after “`xymatrix`”. `groups` should be a set of logical groups. The `hints` will be passed directly to the constructor of [DiagramGrid](#) (page 1745).

For more information about the arguments, see the docstrings of [DiagramGrid](#) (page 1745) and [XypicDiagramDrawer.draw](#).

See also:

[XypicDiagramDrawer](#) (page 1749), [DiagramGrid](#) (page 1745)

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import xypic_draw_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
>>> print(xypic_draw_diagram(diagram))
\xymatrix{
A \ar[d]_{g \circ f} \ar[r]^{f} & B \ar[l]^{g} \\
C & \\
}
```

```
sympy.categories.diagram_drawing.preview_diagram(diagram, masked=None,
                                                  diagram_format="",
                                                  groups=None, output='png',
                                                  viewer=None, euler=True,
                                                  **hints)
```

Combines the functionality of `xypic_draw_diagram` and `sympy.printing.preview`. The arguments `masked`, `diagram_format`, `groups`, and `hints` are passed to `xypic_draw_diagram`, while `output`, `viewer`, and `euler` are passed to `preview`.

See also:

`xypic_diagram_drawer`

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import preview_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> preview_diagram(d)
```

5.41 Differential Geometry Module

5.41.1 Introduction

5.41.2 Base Class Reference

class `sympy.diffgeom.Manifold`

Object representing a mathematical manifold.

The only role that this object plays is to keep a list of all patches defined on the manifold. It does not provide any means to study the topological characteristics of the manifold that it represents.

class sympy.diffgeom.Patch

Object representing a patch on a manifold.

On a manifold one can have many patches that do not always include the whole manifold. On these patches coordinate charts can be defined that permit the parameterization of any point on the patch in terms of a tuple of real numbers (the coordinates).

This object serves as a container/parent for all coordinate system charts that can be defined on the patch it represents.

Examples

Define a Manifold and a Patch on that Manifold:

```
>>> from sympy.diffgeom import Manifold, Patch
>>> m = Manifold('M', 3)
>>> p = Patch('P', m)
>>> p in m.patches
True
```

class sympy.diffgeom.CoordSystem

Contains all coordinate transformation logic.

Examples

Define a Manifold and a Patch, and then define two coord systems on that patch:

```
>>> from sympy import symbols, sin, cos, pi
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> from sympy.simplify import simplify
>>> r, theta = symbols('r, theta')
>>> m = Manifold('M', 2)
>>> patch = Patch('P', m)
>>> rect = CoordSystem('rect', patch)
>>> polar = CoordSystem('polar', patch)
>>> rect in patch.coord_systems
True
```

Connect the coordinate systems. An inverse transformation is automatically found by solve when possible:

```
>>> polar.connect_to(rect, [r, theta], [r*cos(theta), r*sin(theta)])
>>> polar.coord_tuple_transform_to(rect, [0, 2])
Matrix([
[0],
[0]])
>>> polar.coord_tuple_transform_to(rect, [2, pi/2])
Matrix([
[0],
[2]])
>>> rect.coord_tuple_transform_to(polar, [1, 1]).applyfunc(simplify)
Matrix([
[sqrt(2)],
[pi/4]])
```

Calculate the jacobian of the polar to cartesian transformation:


```
>>> polar.jacobian(rect, [r, theta])
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta),  r*cos(theta)]])
```

Define a point using coordinates in one of the coordinate systems:

```
>>> p = polar.point([1, 3*pi/4])
>>> rect.point_to_coords(p)
Matrix([
[-sqrt(2)/2],
[ sqrt(2)/2]])
```

Define a basis scalar field (i.e. a coordinate function), that takes a point and returns its coordinates. It is an instance of `BaseScalarField`.

```
>>> rect.coord_function(0)(p)
-sqrt(2)/2
>>> rect.coord_function(1)(p)
sqrt(2)/2
```

Define a basis vector field (i.e. a unit vector field along the coordinate line). Vectors are also differential operators on scalar fields. It is an instance of `BaseVectorField`.

```
>>> v_x = rect.base_vector(0)
>>> x = rect.coord_function(0)
>>> v_x(x)
1
>>> v_x(v_x(x))
0
```

Define a basis oneform field:

```
>>> dx = rect.base_oneform(0)
>>> dx(v_x)
1
```

If you provide a list of names the fields will print nicely: - without provided names:

```
>>> x, v_x, dx
(rect_0, e_rect_0, drect_0)
```

- with provided names

```
>>> rect = CoordSystem('rect', patch, ['x', 'y'])
>>> rect.coord_function(0), rect.base_vector(0), rect.base_oneform(0)
(x, e_x, dx)
```

base_oneform(coord_index)

Return a basis 1-form field.

The basis one-form field for this coordinate system. It is also an operator on vector fields.

See the docstring of `CoordSystem` for examples.

base_oneforms()

Returns a list of all base oneforms.

For more details see the `base_oneform` method of this class.

base_vector(coord_index)

Return a basis vector field.

The basis vector field for this coordinate system. It is also an operator on scalar fields.

See the docstring of `CoordSystem` for examples.

base_vectors()

Returns a list of all base vectors.

For more details see the `base_vector` method of this class.

connect_to(to_sys, from_coords, to_exprs, inverse=True, fill_in_gaps=False)

Register the transformation used to switch to another coordinate system.

Parameters to_sys

another instance of `CoordSystem`

from_coords

list of symbols in terms of which `to_exprs` is given

to_exprs

list of the expressions of the new coordinate tuple

inverse

try to deduce and register the inverse transformation

fill_in_gaps

try to deduce other transformation that are made possible by composing the present transformation with other already registered transformation

coord_function(coord_index)

Return a `BaseScalarField` that takes a point and returns one of the coords.

Takes a point and returns its coordinate in this coordinate system.

See the docstring of `CoordSystem` for examples.

coord_functions()

Returns a list of all coordinate functions.

For more details see the `coord_function` method of this class.

coord_tuple_transform_to(to_sys, coords)

Transform coords to coord system `to_sys`.

See the docstring of `CoordSystem` for examples.

jacobian(to_sys, coords)

Return the jacobian matrix of a transformation.

point(coords)

Create a `Point` with coordinates given in this coord system.

See the docstring of `CoordSystem` for examples.

point_to_coords(point)

Calculate the coordinates of a point in this coord system.

See the docstring of `CoordSystem` for examples.

class sympy.diffgeom.Point(coord_sys, coords)

Point in a Manifold object.

To define a point you must supply coordinates and a coordinate system.

The usage of this object after its definition is independent of the coordinate system that was used in order to define it, however due to limitations in the simplification routines you can arrive at complicated expressions if you use inappropriate coordinate systems.

Examples

Define the boilerplate Manifold, Patch and coordinate systems:

```
>>> from sympy import symbols, sin, cos, pi
>>> from sympy.diffgeom import (
...     Manifold, Patch, CoordSystem, Point)
>>> r, theta = symbols('r, theta')
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> rect = CoordSystem('rect', p)
>>> polar = CoordSystem('polar', p)
>>> polar.connect_to(rect, [r, theta], [r*cos(theta), r*sin(theta)])
```

Define a point using coordinates from one of the coordinate systems:

```
>>> p = Point(polar, [r, 3*pi/4])
>>> p.coords()
Matrix([
[      r],
[3*pi/4]])
>>> p.coords(rect)
Matrix([
[-sqrt(2)*r/2],
[ sqrt(2)*r/2]])
```

coords(to_sys=None)

Coordinates of the point in a given coordinate system.

If to_sys is None it returns the coordinates in the system in which the point was defined.

class sympy.diffgeom.BaseScalarField

Base Scalar Field over a Manifold for a given Coordinate System.

A scalar field takes a point as an argument and returns a scalar.

A base scalar field of a coordinate system takes a point and returns one of the coordinates of that point in the coordinate system in question.

To define a scalar field you need to choose the coordinate system and the index of the coordinate.

The use of the scalar field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use inappropriate coordinate systems.

You can build complicated scalar fields by just building up SymPy expressions containing BaseScalarField instances.

Examples

Define boilerplate Manifold, Patch and coordinate systems:

```
>>> from sympy import symbols, sin, cos, pi, Function
>>> from sympy.diffgeom import (
...     Manifold, Patch, CoordSystem, Point, BaseScalarField)
>>> r0, theta0 = symbols('r0, theta0')
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> rect = CoordSystem('rect', p)
>>> polar = CoordSystem('polar', p)
>>> polar.connect_to(rect, [r0, theta0], [r0*cos(theta0), r0*sin(theta0)])
```

Point to be used as an argument for the field:

```
>>> point = polar.point([r0, 0])
```

Examples of fields:

```
>>> fx = BaseScalarField(rect, 0)
>>> fy = BaseScalarField(rect, 1)
>>> (fx**2+fy**2).rcall(point)
r0**2
```

```
>>> g = Function('g')
>>> ftheta = BaseScalarField(polar, 1)
>>> fg = g(ftheta-pi)
>>> fg.rcall(point)
g(-pi)
```

class sympy.diffgeom.BaseVectorField

Vector Field over a Manifold.

A vector field is an operator taking a scalar field and returning a directional derivative (which is also a scalar field).

A base vector field is the same type of operator, however the derivation is specifically done with respect to a chosen coordinate.

To define a base vector field you need to choose the coordinate system and the index of the coordinate.

The use of the vector field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use inappropriate coordinate systems.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy import symbols, pi, Function
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import BaseVectorField
>>> from sympy import pprint
>>> x0, y0, r0, theta0 = symbols('x0, y0, r0, theta0')
```

Points to be used as arguments for the field:

```
>>> point_p = R2_p.point([r0, theta0])
>>> point_r = R2_r.point([x0, y0])
```

Scalar field to operate on:

```
>>> g = Function('g')
>>> s_field = g(R2.x, R2.y)
>>> s_field.rcall(point_r)
g(x0, y0)
>>> s_field.rcall(point_p)
g(r0*cos(theta0), r0*sin(theta0))
```

Vector field:

```
>>> v = BaseVectorField(R2_r, 1)
>>> pprint(v(s_field))
/ d      \
|------(g(x, xi_2))|
\dx_i_2      /xi_2=y
>>> pprint(v(s_field).rcall(point_r).doit())
d
---(g(x0, y0))
dy0
>>> pprint(v(s_field).rcall(point_p).doit())
/ d      \
|------(g(r0*cos(theta0), xi_2))|
\dx_i_2      /xi_2=r0*sin(theta0)
```

class sympy.diffgeom.Commutator(v1, v2)

Commutator of two vector fields.

The commutator of two vector fields v_1 and v_2 is defined as the vector field $[v_1, v_2]$ that evaluated on each scalar field f is equal to $v_1(v_2(f)) - v_2(v_1(f))$.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import Commutator
>>> from sympy import pprint
>>> from sympy.simplify import simplify
```

Vector fields:

```
>>> e_x, e_y, e_r = R2.e_x, R2.e_y, R2.e_r
>>> c_xy = Commutator(e_x, e_y)
>>> c_xr = Commutator(e_x, e_r)
>>> c_xy
0
```

Unfortunately, the current code is not able to compute everything:

```
>>> c_xr
Commutator(e_x, e_r)
```

```
>>> simplify(c_xr(R2.y**2).doit())
-2*y**2*cos(theta)/(x**2 + y**2)
```

class sympy.diffgeom.Differential(form_field)

Return the differential (exterior derivative) of a form field.

The differential of a form (i.e. the exterior derivative) has a complicated definition in the general case.

The differential df of the 0-form f is defined for any vector field v as $df(v) = v(f)$.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import Differential
>>> from sympy import pprint
```

Scalar field (0-forms):

```
>>> g = Function('g')
>>> s_field = g(R2.x, R2.y)
```

Vector fields:

```
>>> e_x, e_y, = R2.e_x, R2.e_y
```

Differentials:

```
>>> dg = Differential(s_field)
>>> dg
d(g(x, y))
>>> pprint(dg(e_x))
/ d \
|------(g(xi_1, y))|
\dx_1 / |xi_1=x
>>> pprint(dg(e_y))
/ d \
|------(g(x, xi_2))|
\dx_2 / |xi_2=y
```

Applying the exterior derivative operator twice always results in:

```
>>> Differential(dg)
0
```

class sympy.diffgeom.TensorProduct(*args)

Tensor product of forms.

The tensor product permits the creation of multilinear functionals (i.e. higher order tensors) out of lower order fields (e.g. 1-forms and vector fields). However, the higher tensors thus created lack the interesting features provided by the other type of product, the wedge product, namely they are not antisymmetric and hence are not form fields.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import TensorProduct

>>> TensorProduct(R2.dx, R2.dy)(R2.e_x, R2.e_y)
1
>>> TensorProduct(R2.dx, R2.dy)(R2.e_y, R2.e_x)
0
>>> TensorProduct(R2.dx, R2.x*R2.dy)(R2.x*R2.e_x, R2.e_y)
x**2
>>> TensorProduct(R2.e_x, R2.e_y)(R2.x**2, R2.y**2)
4*x*y
>>> TensorProduct(R2.e_y, R2.dx)(R2.y)
dx
```

You can nest tensor products.

```
>>> tp1 = TensorProduct(R2.dx, R2.dy)
>>> TensorProduct(tp1, R2.dx)(R2.e_x, R2.e_y, R2.e_x)
1
```

You can make partial contraction for instance when ‘raising an index’. Putting None in the second argument of rcall means that the respective position in the tensor product is left as it is.

```
>>> TP = TensorProduct
>>> metric = TP(R2.dx, R2.dx) + 3*TP(R2.dy, R2.dy)
>>> metric.rcall(R2.e_y, None)
3*dy
```

Or automatically pad the args with None without specifying them.

```
>>> metric.rcall(R2.e_y)
3*dy
```

class sympy.diffgeom.WedgeProduct(*args)

Wedge product of forms.

In the context of integration only completely antisymmetric forms make sense. The wedge product permits the creation of such forms.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import WedgeProduct

>>> WedgeProduct(R2.dx, R2.dy)(R2.e_x, R2.e_y)
1
>>> WedgeProduct(R2.dx, R2.dy)(R2.e_y, R2.e_x)
-1
```

(continues on next page)

(continued from previous page)

```
>>> WedgeProduct(R2.dx, R2.x*R2.dy)(R2.x*R2.e_x, R2.e_y)
x**2
>>> WedgeProduct(R2.e_x, R2.e_y)(R2.y, None)
-e_x
```

You can nest wedge products.

```
>>> wp1 = WedgeProduct(R2.dx, R2.dy)
>>> WedgeProduct(wp1, R2.dx)(R2.e_x, R2.e_y, R2.e_x)
0
```

class `sympy.diffgeom.LieDerivative(v_field, expr)`

Lie derivative with respect to a vector field.

The transport operator that defines the Lie derivative is the pushforward of the field to be derived along the integral curve of the field with respect to which one derives.

Examples

```
>>> from sympy.diffgeom import (LieDerivative, TensorProduct)
>>> from sympy.diffgeom.rn import R2
>>> LieDerivative(R2.e_x, R2.y)
0
>>> LieDerivative(R2.e_x, R2.x)
1
>>> LieDerivative(R2.e_x, R2.e_x)
0
```

The Lie derivative of a tensor field by another tensor field is equal to their commutator:

```
>>> LieDerivative(R2.e_x, R2.e_r)
Commutator(e_x, e_r)
>>> LieDerivative(R2.e_x + R2.e_y, R2.x)
1
>>> tp = TensorProduct(R2.dx, R2.dy)
>>> LieDerivative(R2.e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
>>> LieDerivative(R2.e_x, tp).doit()
LieDerivative(e_x, TensorProduct(dx, dy))
```

class `sympy.diffgeom.BaseCovarDerivativeOp(coord_sys, index, christoffel)`

Covariant derivative operator with respect to a base vector.

Examples

```
>>> from sympy.diffgeom.rn import R2, R2_r
>>> from sympy.diffgeom import BaseCovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> ch = metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = BaseCovarDerivativeOp(R2_r, 0, ch)
```

(continues on next page)

(continued from previous page)

```
>>> cvd(R2.x)
1
>>> cvd(R2.x*R2.e_x)
e_x
```

class `sympy.diffgeom.CovarDerivativeOp(wrt, christoffel)`
Covariant derivative operator.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import CovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> ch = metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = CovarDerivativeOp(R2.x*R2.e_x, ch)
>>> cvd(R2.x)
x
>>> cvd(R2.x*R2.e_x)
x*e_x
```

`sympy.diffgeom.intcurve_series(vector_field, param, start_point, n=6, coord_sys=None, coeffs=False)`

Return the series expansion for an integral curve of the field.

Integral curve is a function γ taking a parameter in R to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt} f(\gamma(t))$$

where the given `vector_field` is denoted as V . This holds for any value t for the parameter and any scalar field f .

This equation can also be decomposed of a basis of coordinate functions

$$V(f_i)(\gamma(t)) = \frac{d}{dt} f_i(\gamma(t)) \quad \forall i$$

This function returns a series expansion of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

Parameters `vector_field`

the vector field for which an integral curve will be given

`param`

the argument of the function γ from R to the curve

`start_point`

the point which corresponds to $\gamma(0)$

`n`

the order to which to expand

`coord_sys`

the coordinate system in which to expand coeffs (default False) - if True return a list of elements of the expansion

See also:

[intcurve_diff](#) (page 1765)

Examples

Use the predefined R2 manifold:

```
>>> from sympy.abc import t, x, y
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_series
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([x, y])
>>> vector_field = R2_r.e_x
```

Calculate the series:

```
>>> intcurve_series(vector_field, t, start_point, n=3)
Matrix([
[t + x],
[      y]])
```

Or get the elements of the expansion in a list:

```
>>> series = intcurve_series(vector_field, t, start_point, n=3, coeffs=True)
>>> series[0]
Matrix([
[x],
[y]])
>>> series[1]
Matrix([
[t],
[0]])
>>> series[2]
Matrix([
[0],
[0]])
```

The series in the polar coordinate system:

```
>>> series = intcurve_series(vector_field, t, start_point,
...                           n=3, coord_sys=R2_p, coeffs=True)
>>> series[0]
Matrix([
[sqrt(x**2 + y**2)],
[      atan2(y, x)]]
>>> series[1]
Matrix([
[t*x/sqrt(x**2 + y**2)],
[ -t*y/(x**2 + y**2)]]
>>> series[2]
Matrix([
```

(continues on next page)

(continued from previous page)

```
[t**2*(-x**2/(x**2 + y**2)**(3/2) + 1/sqrt(x**2 + y**2))/2],
[
    t**2*x*y/(x**2 + y**2)**2]])
```

`sympy.diffgeom.intcurve_diffequ(vector_field, param, start_point, coord_sys=None)`

Return the differential equation for an integral curve of the field.

Integral curve is a function γ taking a parameter in R to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt}f(\gamma(t))$$

where the given `vector_field` is denoted as V . This holds for any value t for the parameter and any scalar field f .

This function returns the differential equation of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

Parameters

vector_field the vector field for which an integral curve will be given

param

the argument of the function γ from R to the curve

start_point

the point which corresponds to $\gamma(0)$

coord_sys

the coordinate system in which to give the equations

Returns a tuple of (equations, initial conditions)

See also:

[intcurve_series](#) (page 1763)

Examples

Use the predefined R^2 manifold:

```
>>> from sympy.abc import t
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_diffequ
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([0, 1])
>>> vector_field = -R2.y*R2.e_x + R2.x*R2.e_y
```

Get the equation:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point)
>>> equations
[f_1(t) + Derivative(f_0(t), t), -f_0(t) + Derivative(f_1(t), t)]
>>> init_cond
[f_0(0), f_1(0) - 1]
```

The series in the polar coordinate system:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point, R2_p)
>>> equations
[Derivative(f_0(t), t), Derivative(f_1(t), t) - 1]
>>> init_cond
[f_0(0) - 1, f_1(0) - pi/2]
```

`sympy.diffgeom.vectors_in_basis(expr, to_sys)`

Transform all base vectors in base vectors of a specified coord basis.

While the new base vectors are in the new coordinate system basis, any coefficients are kept in the old system.

Examples

```
>>> from sympy.diffgeom import vectors_in_basis
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> vectors_in_basis(R2_r.e_x, R2_p)
x*e_r/sqrt(x**2 + y**2) - y*e_theta/(x**2 + y**2)
>>> vectors_in_basis(R2_p.e_r, R2_r)
sin(theta)*e_y + cos(theta)*e_x
```

`sympy.diffgeom.twoform_to_matrix(expr)`

Return the matrix representing the twoform.

For the twoform w return the matrix M such that $M[i,j] = w(e_i, e_j)$, where e_i is the i -th base vector field for the coordinate system in which the expression of w is given.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import twoform_to_matrix, TensorProduct
>>> TP = TensorProduct
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[1, 0],
[0, 1]])
>>> twoform_to_matrix(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[x, 0],
[0, 1]])
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy) - TP(R2.dx, R2.dy)/2)
Matrix([
[ 1, 0],
[-1/2, 1]])
```

`sympy.diffgeom.metric_to_Christoffel_1st(expr)`

Return the nested list of Christoffel symbols for the given metric.

This returns the Christoffel symbol of first kind that represents the Levi-Civita connection for the given metric.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_1st, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Christoffel_1st(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> metric_to_Christoffel_1st(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/2, 0], [0, 0]], [[0, 0], [0, 0]]]
```

`sympy.diffgeom.metric_to_Christoffel_2nd(expr)`

Return the nested list of Christoffel symbols for the given metric.

This returns the Christoffel symbol of second kind that represents the Levi-Civita connection for the given metric.

Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> metric_to_Christoffel_2nd(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/(2*x), 0], [0, 0]], [[0, 0], [0, 0]]]
```

`sympy.diffgeom.metric_to_Riemann_components(expr)`

Return the components of the Riemann tensor expressed in a given basis.

Given a metric it calculates the components of the Riemann tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```
>>> from sympy import exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Riemann_components, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Riemann_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

```
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) + R2.r**2*TP(R2.
↳ dtheta, R2.dtheta)
>>> non_trivial_metric
r**2*TensorProduct(dtheta, dtheta) + exp(2*r)*TensorProduct(dr, dr)
>>> riemann = metric_to_Riemann_components(non_trivial_metric)
>>> riemann[0, :, :, :]
[[[0, 0], [0, 0]], [[0, r*exp(-2*r)], [-r*exp(-2*r), 0]]]
>>> riemann[1, :, :, :]
[[[0, -1/r], [1/r, 0]], [[0, 0], [0, 0]]]
```

`sympy.diffgeom.metric_to_Ricci_components(expr)`

Return the components of the Ricci tensor expressed in a given basis.

Given a metric it calculates the components of the Ricci tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```
>>> from sympy import exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Ricci_components, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Ricci_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[0, 0], [0, 0]]
```

```
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +
↪ R2.r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
r**2*TensorProduct(dtheta, dtheta) + exp(2*r)*TensorProduct(dr, dr)
>>> metric_to_Ricci_components(non_trivial_metric)
[[1/r, 0], [0, r*exp(-2*r)]]
```

5.42 Vector Module

The vector module provides tools for basic vector math and differential calculus with respect to 3D Cartesian coordinate systems. This documentation provides an overview of all the features offered, and relevant API.

5.42.1 Guide to Vector

Introduction

This page gives a brief conceptual overview of the functionality present in `sympy.vector`.

Vectors and Scalars

In vector math, we deal with two kinds of quantities – scalars and vectors.

A **scalar** is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A **vector**, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector $\mathbf{V}$, the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\hat{\mathbf{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}})$ m, where , as per standard convention, $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ represent unit vectors along the $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ axes respectively. Therefore, it can be concluded that the distance traveled is $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\|$ m = $5\sqrt{2}$ m. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$.

Coordinate Systems

A **coordinate system** is an abstract mathematical entity used to define the notion of directions and locations in n -dimensional spaces. This module deals with 3-dimensional spaces, with the conventional X , Y and Z axes defined with respect to each coordinate system.

Each coordinate system also has a special reference point called the ‘origin’ defined for it. This point is used either while referring to locations in 3D space, or while calculating the coordinates of pre-defined points with respect to the system.

It is a pretty well-known concept that there is no absolute notion of location or orientation in space. Any given coordinate system defines a unique ‘perspective’ of quantifying positions and directions. Therefore, even if we assume that all systems deal with the same units of measurement, the expression of vectorial and scalar quantities differs according to the coordinate system a certain observer deals with.

Consider two points P and Q in space. Assuming units to be common throughout, the distance between these points remains the same regardless of the coordinate system in which the measurements are being made. However, the 3-D coordinates of each of the two points, as well as the position vector of any of the points with respect to the other, do not. In fact, these two quantities don’t make sense at all, unless they are being measured keeping in mind a certain location and orientation of the measurer (essentially the coordinate system).

Therefore, it is quite clear that the orientation and location (of the origin) of a coordinate system define the way different quantities will be expressed with respect to it. Neither of the two properties can be measured on an absolute scale, but rather with respect to another coordinate system. The orientation of one system with respect to another is measured using the rotation matrix, while the relative position can be quantified via the position vector of one system’s origin with respect to the other.

Fields

A **field** is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). Since we only deal with 3D spaces in this module, a field is defined as a function of the x , y and z coordinates corresponding to a location in the coordinate system. Here, x , y and z act as scalar variables defining the position of a general point.

For example, temperature in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location (x, y, z) of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $\vec{g}(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\hat{i} + 2x^2\hat{j}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Basic Implementation details

Coordinate Systems and Vectors

Currently, `sympy.vector` is able to deal with the Cartesian (also called rectangular), spherical and other curvilinear coordinate systems.

A 3D Cartesian coordinate system can be initialized in `sympy.vector` as

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
```

The string parameter to the constructor denotes the name assigned to the system, and will primarily be used for printing purposes.

Once a coordinate system (in essence, a `CoordSys3D` instance) has been defined, we can access the orthonormal unit vectors (i.e. the $\hat{i}$, $\hat{j}$ and $\hat{k}$ vectors) and coordinate variables/base scalars (i.e. the x , y and z variables) corresponding to it. We will talk about coordinate variables in the later sections.

The basis vectors for the X , Y and Z axes can be accessed using the `i`, `j` and `k` properties respectively.

```
>>> N.i
N.i
>>> type(N.i)
<class 'sympy.vector.vector.BaseVector'>
```

As seen above, the basis vectors are all instances of a class called `BaseVector`.

When a `BaseVector` is multiplied by a scalar (essentially any SymPy Expr), we get a `VectorMul` - the product of a base vector and a scalar.

```
>>> 3*N.i
3*N.i
>>> type(3*N.i)
<class 'sympy.vector.vector.VectorMul'>
```

Addition of `VectorMul` and `BaseVectors` gives rise to formation of `VectorAdd` - except for special cases, ofcourse.

```
>>> v = 2*N.i + N.j
>>> type(v)
<class 'sympy.vector.vector.VectorAdd'>
>>> v - N.j
2*N.i
>>> type(v - N.j)
<class 'sympy.vector.vector.VectorMul'>
```

What about a zero vector? It can be accessed using the `zero` attribute assigned to class `Vector`. Since the notion of a zero vector remains the same regardless of the coordinate system in consideration, we use `Vector.zero` wherever such a quantity is required.

```
>>> from sympy.vector import Vector
>>> Vector.zero
0
>>> type(Vector.zero)
<class 'sympy.vector.vector.VectorZero'>
```

(continues on next page)

(continued from previous page)

```
>>> N.i + Vector.zero
N.i
>>> Vector.zero == 2*Vector.zero
True
```

All the classes shown above - `BaseVector`, `VectorMul`, `VectorAdd` and `VectorZero` are subclasses of `Vector`.

You should never have to instantiate objects of any of the subclasses of `Vector`. Using the `BaseVector` instances assigned to a `CoordSys3D` instance and (if needed) `Vector.zero` as building blocks, any sort of vectorial expression can be constructed with the basic mathematical operators `+`, `-`, `*`, and `/`.

```
>>> v = N.i - 2*N.j
>>> v/3
1/3*N.i + (-2/3)*N.j
>>> v + N.k
N.i + (-2)*N.j + N.k
>>> Vector.zero/2
0
>>> (v/3)*4
4/3*N.i + (-8/3)*N.j
```

In addition to the elementary mathematical operations, the vector operations of `dot` and `cross` can also be performed on `Vector`.

```
>>> v1 = 2*N.i + 3*N.j - N.k
>>> v2 = N.i - 4*N.j + N.k
>>> v1.dot(v2)
-11
>>> v1.cross(v2)
(-1)*N.i + (-3)*N.j + (-11)*N.k
>>> v2.cross(v1)
N.i + 3*N.j + 11*N.k
```

The `&` and `^` operators have been overloaded for the `dot` and `cross` methods respectively.

```
>>> v1 & v2
-11
>>> v1 ^ v2
(-1)*N.i + (-3)*N.j + (-11)*N.k
```

However, this is not the recommended way of performing these operations. Using the original methods makes the code clearer and easier to follow.

In addition to these operations, it is also possible to compute the outer products of `Vector` instances in `sympy.vector`. More on that in a little bit.

SymPy operations on Vectors

The SymPy operations of `simplify`, `trigsimp`, `diff`, and `factor` work on `Vector` objects, with the standard SymPy API.

In essence, the methods work on the measure numbers (The coefficients of the basis vectors) present in the provided vectorial expression.

```

>>> from sympy.abc import a, b, c
>>> from sympy import sin, cos, trigsimp, diff
>>> v = (a*b + a*c + b**2 + b*c)*N.i + N.j
>>> v.factor()
((a + b)*(b + c))*N.i + N.j
>>> v = (sin(a)**2 + cos(a)**2)*N.i - (2*cos(b)**2 - 1)*N.k
>>> trigsimp(v)
N.i + (-cos(2*b))*N.k
>>> v.simplify()
N.i + (-cos(2*b))*N.k
>>> diff(v, b)
(4*sin(b)*cos(b))*N.k
>>> from sympy import Derivative
>>> Derivative(v, b).doit()
(4*sin(b)*cos(b))*N.k

```

Integral also works with Vector instances, similar to Derivative.

```

>>> from sympy import Integral
>>> v1 = a*N.i + sin(a)*N.j - N.k
>>> Integral(v1, a)
(Integral(a, a))*N.i + (Integral(sin(a), a))*N.j + (Integral(-1, a))*N.k
>>> Integral(v1, a).doit()
a**2/2*N.i + (-cos(a))*N.j + (-a)*N.k

```

Points

As mentioned before, every coordinate system corresponds to a unique origin point. Points, in general, have been implemented in `sympy.vector` in the form of the `Point` class.

To access the origin of system, use the `origin` property of the `CoordSys3D` class.

```

>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.origin
N.origin
>>> type(N.origin)
<class 'sympy.vector.point.Point'>

```

You can instantiate new points in space using the `locate_new` method of `Point`. The arguments include the name(string) of the new `Point`, and its position vector with respect to the 'parent' `Point`.

```

>>> from sympy.abc import a, b, c
>>> P = N.origin.locate_new('P', a*N.i + b*N.j + c*N.k)
>>> Q = P.locate_new('Q', -b*N.j)

```

Like `Vector`, a user never has to expressly instantiate an object of `Point`. This is because any location in space (albeit relative) can be pointed at by using the origin of a `CoordSys3D` as the reference, and then using `locate_new` on it and subsequent `Point` instances.

The position vector of a `Point` with respect to another `Point` can be computed using the `position_wrt` method.

```

>>> P.position_wrt(Q)
b*N.j

```

(continues on next page)

(continued from previous page)

```
>>> Q.position_wrt(N.origin)
a*N.i + c*N.k
```

Additionally, it is possible to obtain the X , Y and Z coordinates of a `Point` with respect to a `CoordSys3D` in the form of a tuple. This is done using the `express_coordinates` method.

```
>>> Q.express_coordinates(N)
(a, 0, c)
```

Dyadics

A dyadic, or dyadic tensor, is a second-order tensor formed by the juxtaposition of pairs of vectors. Therefore, the outer products of vectors give rise to the formation of dyadics. Dyadic tensors have been implemented in `sympy.vector` in the `Dyadic` class.

Once again, you never have to instantiate objects of `Dyadic`. The outer products of vectors can be computed using the `outer` method of `Vector`. The `|` operator has been overloaded for `outer`.

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
>>> N.i|N.j
(N.i|N.j)
```

Similar to `Vector`, `Dyadic` also has subsequent subclasses like `BaseDyadic`, `DyadicMul`, `DyadicAdd`. As with `Vector`, a zero dyadic can be accessed from `Dyadic.zero`.

All basic mathematical operations work with `Dyadic` too.

```
>>> dyad = N.i.outer(N.k)
>>> dyad*3
3*(N.i|N.k)
>>> dyad - dyad
0
>>> dyad + 2*(N.j|N.i)
(N.i|N.k) + 2*(N.j|N.i)
```

`dot` and `cross` also work among `Dyadic` instances as well as between a `Dyadic` and `Vector` (and also vice versa) - as per the respective mathematical definitions. As with `Vector`, `&` and `^` have been overloaded for `dot` and `cross`.

```
>>> d = N.i.outer(N.j)
>>> d.dot(N.j|N.j)
(N.i|N.j)
>>> d.dot(N.i)
0
>>> d.dot(N.j)
N.i
>>> N.i.dot(d)
N.j
>>> N.k ^ d
(N.j|N.j)
```

More about Coordinate Systems

We will now look at how we can initialize new coordinate systems in `sympy.vector`, transformed in user-defined ways with respect to already-existing systems.

Locating new systems

We already know that the `origin` property of a `CoordSys3D` corresponds to the `Point` instance denoting its origin reference point.

Consider a coordinate system N . Suppose we want to define a new system M , whose origin is located at $3\hat{i} + 4\hat{j} + 5\hat{k}$ from N 's origin. In other words, the coordinates of M 's origin from N 's perspective happen to be $(3, 4, 5)$. Moreover, this would also mean that the coordinates of N 's origin with respect to M would be $(-3, -4, -5)$.

This can be achieved programmatically as follows -

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> M = N.locate_new('M', 3*N.i + 4*N.j + 5*N.k)
>>> M.position_wrt(N)
3*N.i + 4*N.j + 5*N.k
>>> N.origin.express_coordinates(M)
(-3, -4, -5)
```

It is worth noting that M 's orientation is the same as that of N . This means that the rotation matrix of N with respect to M , and also vice versa, is equal to the identity matrix of dimensions 3×3 . The `locate_new` method initializes a `CoordSys3D` that is only translated in space, not re-oriented, relative to the 'parent' system.

Orienting new systems

Similar to 'locating' new systems, `sympy.vector` also allows for initialization of new `CoordSys3D` instances that are oriented in user-defined ways with respect to existing systems.

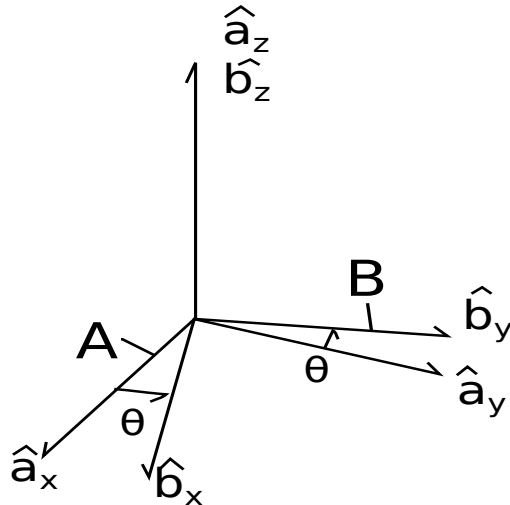
Suppose you have a coordinate system A .

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
```

You want to initialize a new coordinate system B , that is rotated with respect to A 's Z-axis by an angle θ .

```
>>> from sympy import Symbol
>>> theta = Symbol('theta')
```

The orientation is shown in the diagram below:



There are two ways to achieve this.

Using a method of CoordSys3D directly

This is the easiest, cleanest, and hence the recommended way of doing it.

```
>>> B = A.orient_new_axis('B', theta, A.k)
```

This initializes B with the required orientation information with respect to A .

CoordSys3D provides the following direct orientation methods in its API-

1. orient_new_axis
2. orient_new_body
3. orient_new_space
4. orient_new_quaternion

Please look at the CoordSys3D class API given in the docs of this module, to know their functionality and required arguments in detail.

Using Orienter(s) and the orient_new method

You would first have to initialize an AxisOrienter instance for storing the rotation information.

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(theta, A.k)
```

And then apply it using the orient_new method, to obtain B .

```
>>> B = A.orient_new('B', axis_orienter)
```

orient_new also lets you orient new systems using multiple Orienter instances, provided in an iterable. The rotations/orientations are applied to the new system in the order the Orienter instances appear in the iterable.

```
>>> from sympy.vector import BodyOrienter
>>> from sympy.abc import a, b, c
>>> body_orienter = BodyOrienter(a, b, c, 'XYZ')
>>> C = A.orient_new('C', (axis_orienter, body_orienter))
```

The `sympy.vector` API provides the following four `Orienter` classes for orientation purposes-

1. `AxisOrienter`
2. `BodyOrienter`
3. `SpaceOrienter`
4. `QuaternionOrienter`

Please refer to the API of the respective classes in the docs of this module to know more.

In each of the above examples, the origin of the new coordinate system coincides with the origin of the 'parent' system.

```
>>> B.position_wrt(A)
0
```

To compute the rotation matrix of any coordinate system with respect to another one, use the `rotation_matrix` method.

```
>>> B = A.orient_new_axis('B', a, A.k)
>>> B.rotation_matrix(A)
Matrix([
[ cos(a), sin(a), 0],
[-sin(a), cos(a), 0],
[      0,      0, 1]])
>>> B.rotation_matrix(B)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Orienting AND Locating new systems

What if you want to initialize a new system that is not only oriented in a pre-defined way, but also translated with respect to the parent?

Each of the `orient_new_<method of orientation>` methods, as well as the `orient_new` method, support a `location` keyword argument.

If a `Vector` is supplied as the value for this `kwarg`, the new system's origin is automatically defined to be located at that position vector with respect to the parent coordinate system.

Thus, the orientation methods also act as methods to support orientation+ location of the new systems.

```
>>> C = A.orient_new_axis('C', a, A.k, location=2*A.j)
>>> C.position_wrt(A)
2*A.j
>>> from sympy.vector import express
>>> express(A.position_wrt(C), C)
(-2*sin(a))*C.i + (-2*cos(a))*C.j
```

More on the `express` function in a bit.

Transforming new system

The most general way of creating user-defined system is to use transformation parameter in `CoordSys3D`. Here we can define any transformation equations. If we are interested in some typical curvilinear coordinate system different that Cartesian, we can also use some predefined ones. It could be also possible to translate or rotate system by setting appropriate transformation equations.

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import sin, cos
>>> A = CoordSys3D('A', transformation='spherical')
>>> B = CoordSys3D('A', transformation=lambda x,y,z: (x*sin(y), x*cos(y), z))
```

In `CoordSys3D` is also dedicated method, `create_new` which works similarly to methods like `locate_new`, `orient_new_axis` etc.

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.create_new('B', transformation='spherical')
```

Expression of quantities in different coordinate systems

Vectors and Dyadics

As mentioned earlier, the same vector attains different expressions in different coordinate systems. In general, the same is true for scalar expressions and dyadic tensors.

`sympy.vector` supports the expression of vector/scalar quantities in different coordinate systems using the `express` function.

For purposes of this section, assume the following initializations-

```
>>> from sympy.vector import CoordSys3D, express
>>> from sympy.abc import a, b, c
>>> N = CoordSys3D('N')
>>> M = N.orient_new_axis('M', a, N.k)
```

Vector instances can be expressed in user defined systems using `express`.

```
>>> v1 = N.i + N.j + N.k
>>> express(v1, M)
(sin(a) + cos(a))*M.i + (-sin(a) + cos(a))*M.j + M.k
>>> v2 = N.i + M.j
>>> express(v2, N)
(-sin(a) + 1)*N.i + (cos(a))*N.j
```

Apart from Vector instances, `express` also supports reexpression of scalars (general SymPy Expr) and Dyadic objects.

`express` also accepts a second coordinate system for re-expressing Dyadic instances.

```
>>> d = 2*(M.i | N.j) + 3*(M.j | N.k)
>>> express(d, M)
(2*sin(a))*(M.i|M.i) + (2*cos(a))*(M.i|M.j) + 3*(M.j|M.k)
>>> express(d, M, N)
2*(M.i|N.j) + 3*(M.j|N.k)
```

Coordinate Variables

The location of a coordinate system's origin does not affect the re-expression of `BaseVector` instances. However, it does affect the way `BaseScalar` instances are expressed in different systems.

`BaseScalar` instances, are coordinate 'symbols' meant to denote the variables used in the definition of vector/scalar fields in `sympy.vector`.

For example, consider the scalar field $\mathbf{T}_N(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x} + \mathbf{y} + \mathbf{z}$ defined in system N . Thus, at a point with coordinates (a, b, c) , the value of the field would be $a + b + c$. Now consider system R , whose origin is located at $(1, 2, 3)$ with respect to N (no change of orientation). A point with coordinates (a, b, c) in R has coordinates $(a + 1, b + 2, c + 3)$ in N . Therefore, the expression for $\mathbf{T}_N$ in R becomes $\mathbf{T}_R(x, y, z) = x + y + z + 6$.

Coordinate variables, if present in a vector/scalar/dyadic expression, can also be re-expressed in a given coordinate system, by setting the `variables` keyword argument of `express` to `True`.

The above mentioned example, done programmatically, would look like this -

```
>>> R = N.locate_new('R', N.i + 2*N.j + 3*N.k)
>>> T_N = N.x + N.y + N.z
>>> express(T_N, R, variables=True)
R.x + R.y + R.z + 6
```

Other expression-dependent methods

The `to_matrix` method of `Vector` and `express_coordinates` method of `Point` also return different results depending on the coordinate system being provided.

```
>>> P = R.origin.locate_new('P', a*R.i + b*R.j + c*R.k)
>>> P.express_coordinates(N)
(a + 1, b + 2, c + 3)
>>> P.express_coordinates(R)
(a, b, c)
>>> v = N.i + N.j + N.k
>>> v.to_matrix(M)
Matrix([
[ sin(a) + cos(a)],
[-sin(a) + cos(a)],
[                1]])
>>> v.to_matrix(N)
Matrix([
[1],
[1],
[1]])
```

Scalar and Vector Field Functionality

Implementation in `sympy.vector`

Scalar and vector fields

In `sympy.vector`, every `CoordSysCartesian` instance is assigned basis vectors corresponding to the X , Y and Z axes. These can be accessed using the properties named `i`, `j` and `k`

respectively. Hence, to define a vector $\mathbf{v}$ of the form $3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}$ with respect to a given frame $\mathbf{R}$, you would do

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = 3*R.i + 4*R.j + 5*R.k
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in the earlier section of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented in a special class called `BaseScalar`, and are assigned to every coordinate system, one for each axis from X , Y and Z . These coordinate variables are used to form the expressions of vector or scalar fields in 3D space. For a system R , the X , Y and Z `BaseScalars` instances can be accessed using the $R.x$, $R.y$ and $R.z$ expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
>>> electric_potential
2*R.x**2*R.y
```

It is to be noted that `BaseScalar` instances can be used just like any other SymPy `Symbol`, except that they store the information about the coordinate system and axis they correspond to.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to x (i.e. $R.x$), you would use the `diff` method.

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
>>> from sympy import diff
>>> diff(electric_potential, R.x)
4*R.x*R.y
```

It is worth noting that having a `BaseScalar` in the expression implies that a 'field' changes with position, in 3D space. Technically speaking, a simple `Expr` with no `BaseScalar`s is still a field, though constant.

Like scalar fields, vector fields that vary with position can also be constructed using `BaseScalar`s in the measure-number expressions.

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = R.x**2*R.i + 2*R.x*R.z*R.k
```

The Del operator

The Del, or 'Nabla' operator - written as ∇ is commonly known as the vector differential operator. Depending on its usage in a mathematical expression, it may denote the gradient of a scalar field, the divergence of a vector field, or the curl of a vector field.

Essentially, ∇ is not technically an ‘operator’, but a convenient mathematical notation to denote any one of the aforementioned field operations.

In `sympy.vector`, ∇ has been implemented as the `Del()` class. The instance of this class is independent of coordinate system. Hence, the ∇ operator would be accessible as `Del()`.

Given below is an example of usage of the `Del()` class.

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> gradient_field = delop(C.x*C.y*C.z)
>>> gradient_field
(Derivative(C.x*C.y*C.z, C.x))*C.i + (Derivative(C.x*C.y*C.z, C.y))*C.j
+ (Derivative(C.x*C.y*C.z, C.z))*C.k
```

The above expression can be evaluated using the `SymPy doit()` routine.

```
>>> gradient_field.doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Usage of the ∇ notation in `sympy.vector` has been described in greater detail in the subsequent subsections.

Field operators and related functions

Here we describe some basic field-related functionality implemented in `sympy.vector`.

Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector $\mathbf{F}$, denoted by $\nabla \times \mathbf{F}$ is given by:

$$\nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{\mathbf{i}} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{\mathbf{j}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{\mathbf{k}}$$

where F_x denotes the X component of vector $\mathbf{F}$.

Computing the curl of a vector field in `sympy.vector` can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.cross(C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
>>> (delop ^ C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Or by using the dedicated function

```
>>> from sympy.vector import curl
>>> curl(C.x*C.y*C.z*C.i)
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Divergence

Divergence is a vector operator that measures the magnitude of a vector field's source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector $\mathbf{F}$, denoted by $\nabla \cdot \mathbf{F}$ is given by:

$$\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}$$

where U , V and W denote the X , Y and Z components of $\mathbf{F}$ respectively.

Computing the divergence of a vector field in `sympy.vector` can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.dot(C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
>>> (delop & C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

Or by using the dedicated function

```
>>> from sympy.vector import divergence
>>> divergence(C.x*C.y*C.z*(C.i + C.j + C.k))
C.x*C.y + C.x*C.z + C.y*C.z
```

Gradient

Consider a scalar field $f(x, y, z)$ in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of f with respect to x , y and z in the X , Y and Z axes respectively.

In the 3D Cartesian system, the divergence of a scalar field f , denoted by ∇f is given by -

$$\nabla f = \frac{\partial f}{\partial x} \hat{\mathbf{i}} + \frac{\partial f}{\partial y} \hat{\mathbf{j}} + \frac{\partial f}{\partial z} \hat{\mathbf{k}}$$

Computing the divergence of a vector field in `sympy.vector` can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Or by using the dedicated function

```
>>> from sympy.vector import gradient
>>> gradient(C.x*C.y*C.z)
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Directional Derivative

Apart from the above three common applications of ∇ , it is also possible to compute the directional derivative of a field wrt a `Vector` in `sympy.vector`.

By definition, the directional derivative of a field $\mathbf{F}$ along a vector v at point x represents the instantaneous rate of change of $\mathbf{F}$ moving through x with the velocity v . It is represented mathematically as: $(\vec{v} \cdot \nabla) \mathbf{F}(x)$.

Directional derivatives of vector and scalar fields can be computed in `sympy.vector` using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> vel = C.i + C.j + C.k
>>> scalar_field = C.x*C.y*C.z
>>> vector_field = C.x*C.y*C.z*C.i
>>> (vel.dot(delop))(scalar_field)
C.x*C.y + C.x*C.z + C.y*C.z
>>> (vel & delop)(vector_field)
(C.x*C.y + C.x*C.z + C.y*C.z)*C.i
```

Or by using the dedicated function

```
>>> from sympy.vector import directional_derivative
>>> directional_derivative(C.x*C.y*C.z, 3*C.i + 4*C.j + C.k)
C.x*C.y + 4*C.x*C.z + 3*C.y*C.z
```

Field operator in orthogonal curvilinear coordinate system

`vector` package supports calculation in different kind of orthogonal curvilinear coordinate system. To do that, scaling factor (also known as Lamé coefficients) are used to express curl, divergence or gradient in desired type of coordinate system.

For example if we want to calculate gradient in cylindrical coordinate system all we need to do is to create proper coordinate system

```
>>> from sympy.vector import CoordSys3D
>>> c = CoordSys3D('c', transformation='cylindrical', variable_names=(" $r$ ", " $\theta$ ", " $z$ "))
>>> gradient(c.r*c.theta*c.z)
c.theta*c.z*c.i + c.z*c.j + c.r*c.theta*c.k
```

Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path travelled. A conservative vector field is also said to be 'irrotational', since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in `sympy.vector`, the `is_conservative` function can be used.

```
>>> from sympy.vector import CoordSys3D, is_conservative
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_conservative(field)
True
>>> curl(field)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in `sympy.vector`, the `is_solenoidal` function can be used.

```
>>> from sympy.vector import CoordSys3D, is_solenoidal
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_solenoidal(field)
True
>>> divergence(field)
0
```

Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the 'scalar potential field' corresponding to the aforementioned conservative field.

The `scalar_potential` function in `sympy.vector` calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```
>>> from sympy.vector import CoordSys3D, scalar_potential
>>> R = CoordSys3D('R')
>>> conservative_field = 4*R.x*R.y*R.z*R.i + 2*R.x**2*R.z*R.j + 2*R.x**2*R.y*R.k
>>> scalar_potential(conservative_field, R)
2*R.x**2*R.y*R.z
```

Providing a non-conservative vector field as an argument to `scalar_potential` raises a `ValueError`.

The scalar potential difference, or simply 'potential difference', corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in `sympy.vector`.

```
>>> from sympy.vector import CoordSys3D, Point
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', 1*R.i + 2*R.j + 3*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
4
```

If provided with a scalar expression instead of a vector field, `scalar_potential_difference` returns the difference between the values of that scalar field at the two given points in space.

General examples of usage

This section details the solution of two basic problems in vector math/calculus using the `sympy.vector` package.

Quadrilateral problem

The Problem

OABC is any quadrilateral in 3D space. P is the midpoint of OA, Q is the midpoint of AB, R is the midpoint of BC and S is the midpoint of OC. Prove that PQ is parallel to SR

Solution

The solution to this problem demonstrates the usage of `Point`, and basic operations on `Vector`.

Define a coordinate system

```
>>> from sympy.vector import CoordSys3D
>>> Sys = CoordSys3D('Sys')
```

Define point O to be Sys' origin. We can do this without loss of generality

```
>>> O = Sys.origin
```

Define point A with respect to O

```
>>> from sympy import symbols
>>> a1, a2, a3 = symbols('a1 a2 a3')
>>> A = O.locate_new('A', a1*Sys.i + a2*Sys.j + a3*Sys.k)
```

Similarly define points B and C

```
>>> b1, b2, b3 = symbols('b1 b2 b3')
>>> B = O.locate_new('B', b1*Sys.i + b2*Sys.j + b3*Sys.k)
>>> c1, c2, c3 = symbols('c1 c2 c3')
>>> C = O.locate_new('C', c1*Sys.i + c2*Sys.j + c3*Sys.k)
```

P is the midpoint of OA. Lets locate it with respect to O (you could also define it with respect to A).

```
>>> P = O.locate_new('P', A.position_wrt(O) + (O.position_wrt(A) / 2))
```

Similarly define points Q, R and S as per the problem definitions.

```
>>> Q = A.locate_new('Q', B.position_wrt(A) / 2)
>>> R = B.locate_new('R', C.position_wrt(B) / 2)
>>> S = O.locate_new('R', C.position_wrt(O) / 2)
```

Now compute the vectors in the directions specified by PQ and SR.

```
>>> PQ = Q.position_wrt(P)
>>> SR = R.position_wrt(S)
```

Compute cross product

```
>>> PQ.cross(SR)
0
```

Since the cross product is a zero vector, the two vectors have to be parallel, thus proving that $PQ \parallel SR$.

Third product rule for Del operator

See

The Problem

Prove the third rule - $\nabla \cdot (f\vec{v}) = f(\nabla \cdot \vec{v}) + \vec{v} \cdot (\nabla f)$

Solution

Start with a coordinate system

```
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
```

The scalar field f and the measure numbers of the vector field $\vec{v}$ are all functions of the coordinate variables of the coordinate system in general. Hence, define SymPy functions that way.

```
>>> from sympy import symbols
>>> v1, v2, v3, f = symbols('v1 v2 v3 f', type="Function")
```

v_1, v_2 and v_3 are the X, Y and Z components of the vector field respectively.

Define the vector field as $vfield$ and the scalar field as $sfield$.

```
>>> vfield = v1(C.x, C.y, C.z)*C.i + v2(C.x, C.y, C.z)*C.j + v3(C.x, C.y, C.z)*C.k
>>> ffield = f(C.x, C.y, C.z)
```

Construct the expression for the LHS of the equation using `Del()`.

```
>>> lhs = (delop.dot(ffield * vfield)).doit()
```

Similarly, the RHS would be defined.

```
>>> rhs = ((vfield.dot(delop(ffield))) + (ffield * (delop.dot(vfield)))).doit()
```

Now, to prove the product rule, we would just need to equate the expanded and simplified versions of the lhs and the rhs, so that the SymPy expressions match.

```
>>> lhs.expand().simplify() == rhs.expand().doit().simplify()
True
```

Thus, the general form of the third product rule mentioned above can be proven using `sympy.vector`.

5.42.2 Vector API

Essential Classes in `sympy.vector` (docstrings)

`CoordSys3D`

```
class sympy.vector.coordsysrect.CoordSys3D(name, location=None, rotation_matrix=None, parent=None, vector_names=None, variable_names=None, latex_vects=None, pretty_vects=None, latex_scalars=None, pretty_scalars=None, transformation=None)
```

Represents a coordinate system in 3-D space.

```
__init__(name, location=None, rotation_matrix=None, parent=None, vector_names=None, variable_names=None, latex_vects=None, pretty_vects=None, latex_scalars=None, pretty_scalars=None, transformation=None)
```

The orientation/location parameters are necessary if this system is being defined at a certain orientation or location wrt another.

Parameters **name** : str

The name of the new `CoordSys3D` instance.

transformation : Lambda, Tuple, str

Transformation defined by transformation equations or chosen from predefined ones.

location : Vector

The position vector of the new system's origin wrt the parent instance.

rotation_matrix : SymPy ImmutableMatrix

The rotation matrix of the new coordinate system with respect to the parent. In other words, the output of `new_system.rotation_matrix(parent)`.

parent : `CoordSys3D`

The coordinate system wrt which the orientation/location (or both) is being defined.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

create_new(name, transformation, variable_names=None, vector_names=None)

Returns a `CoordSys3D` which is connected to self by transformation.

Parameters name : str

The name of the new CoordSys3D instance.

transformation : Lambda, Tuple, str

Transformation defined by transformation equations or chosen from predefined ones.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> a = CoordSys3D('a')
>>> b = a.create_new('b', transformation='spherical')
>>> b.transformation_to_parent()
(b.r*sin(b.theta)*cos(b.phi), b.r*sin(b.phi)*sin(b.theta), b.r*cos(b.theta))
>>> b.transformation_from_parent()
(sqrt(a.x**2 + a.y**2 + a.z**2), acos(a.z/sqrt(a.x**2 + a.y**2 + a.z**2)),
atan2(a.y, a.x))
```

locate_new(name, position, vector_names=None, variable_names=None)

Returns a CoordSys3D with its origin located at the given position wrt this coordinate system's origin.

Parameters name : str

The name of the new CoordSys3D instance.

position : Vector

The position vector of the new system's origin wrt this one.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.locate_new('B', 10 * A.i)
>>> B.origin.position_wrt(A.origin)
10*A.i
```

orient_new(name, orienters, location=None, vector_names=None, variable_names=None)

Creates a new CoordSys3D oriented in the user-specified way with respect to this system.

Please refer to the documentation of the orienter classes for more information about the orientation procedure.

Parameters name : str

The name of the new CoordSys3D instance.

orienters : iterable/Orienter

An Orienter or an iterable of Orienters for orienting the new coordinate system. If an Orienter is provided, it is applied to get the new system. If an iterable is provided, the orienters will be applied in the order in which they appear in the iterable.

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
```

Using an AxisOrienter

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> A = N.orient_new('A', (axis_orienter, ))
```

Using a BodyOrienter

```
>>> from sympy.vector import BodyOrienter
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> B = N.orient_new('B', (body_orienter, ))
```

Using a SpaceOrienter

```
>>> from sympy.vector import SpaceOrienter
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> C = N.orient_new('C', (space_orienter, ))
```

Using a QuaternionOrienter

```
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> D = N.orient_new('D', (q_orienter, ))
```

orient_new_axis(name, angle, axis, location=None, vector_names=None, variable_names=None)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

Parameters name : string

The name of the new coordinate system

angle : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_axis('B', q1, N.i + 2 * N.j)
```

orient_new_body(name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see http://en.wikipedia.org/wiki/Euler_angles.

Parameters **name** : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> D = N.orient_new_body('D', q1, q2, q3, '123')
```

is same as

```
>>> D = N.orient_new_axis('D', q1, N.i)
>>> D = D.orient_new_axis('D', q2, D.j)
>>> D = D.orient_new_axis('D', q3, D.k)
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> B = N.orient_new_body('B', q1, q2, q3, '123')
>>> B = N.orient_new_body('B', q1, q2, 0, 'ZXZ')
>>> B = N.orient_new_body('B', 0, 0, 0, 'XYX')
```

orient_new_quaternion(name, q0, q1, q2, q3, location=None, vector_names=None, variable_names=None)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

$q0 = \cos(\theta/2)$

$q1 = \lambda_x \sin(\theta/2)$

$q2 = \lambda_y \sin(\theta/2)$

$q3 = \lambda_z \sin(\theta/2)$

Quaternion does not take in a rotation order.

Parameters **name** : string

The name of the new coordinate system

q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_quaternion('B', q0, q1, q2, q3)
```

orient_new_space(name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters **name** : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

See also:

[**CoordSys3D.orient_new_body** \(page 1789\)](#) method to orient via Euler angles

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> D = N.orient_new_space('D', q1, q2, q3, '312')
```

is same as

```
>>> B = N.orient_new_axis('B', q1, N.i)
>>> C = B.orient_new_axis('C', q2, N.j)
>>> D = C.orient_new_axis('D', q3, N.k)
```

position_wrt(other)

Returns the position vector of the origin of this coordinate system with respect to another Point/CoordSys3D.

Parameters other : Point/CoordSys3D

If other is a Point, the position of this system's origin wrt it is returned. If its an instance of CoordSysRect, the position wrt its origin is returned.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N1 = N.locate_new('N1', 10 * N.i)
>>> N.position_wrt(N1)
(-10)*N.i
```

rotation_matrix(other)

Returns the direction cosine matrix(DCM), also known as the 'rotation matrix' of this coordinate system with respect to another system.

If v_a is a vector defined in system 'A' (in matrix format) and v_b is the same vector defined in system 'B', then $v_a = A.rotation\_matrix(B) * v_b$.

A SymPy Matrix is returned.

Parameters other : CoordSys3D

The system which the DCM is generated to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> A = N.orient_new_axis('A', q1, N.i)
>>> N.rotation_matrix(A)
Matrix([
[1,      0,      0],
[0, cos(q1), -sin(q1)],
[0, sin(q1),  cos(q1)])
```

scalar_map(other)

Returns a dictionary which expresses the coordinate variables (base scalars) of this frame in terms of the variables of otherframe.

Parameters otherframe : CoordSys3D

The other system to map the variables to.

Examples

```

>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol
>>> A = CoordSys3D('A')
>>> q = Symbol('q')
>>> B = A.orient_new_axis('B', q, A.k)
>>> A.scalar_map(B)
{A.x: B.x*cos(q) - B.y*sin(q), A.y: B.x*sin(q) + B.y*cos(q), A.z: B.z}

```

Vector

class sympy.vector.vector.Vector

Super class for all Vector classes. Ideally, neither this class nor any of its subclasses should be instantiated by the user.

components

Returns the components of this vector in the form of a Python dictionary mapping BaseVector instances to the corresponding measure numbers.

Examples

```

>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.components
{C.i: 3, C.j: 4, C.k: 5}

```

cross(other)

Returns the cross product of this Vector with another Vector or Dyadic instance. The cross product is a Vector, if 'other' is a Vector. If 'other' is a Dyadic, this returns a Dyadic instance.

Parameters other: Vector/Dyadic

The Vector or Dyadic we are crossing with.

Examples

```

>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> C.i.cross(C.j)
C.k
>>> C.i ^ C.i
0
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v ^ C.i
5*C.j + (-4)*C.k
>>> d = C.i.outer(C.i)
>>> C.j.cross(d)
(-1)*(C.k|C.i)

```

dot(other)

Returns the dot product of this Vector, either with another Vector, or a Dyadic, or

a Del operator. If 'other' is a Vector, returns the dot product scalar (SymPy expression). If 'other' is a Dyadic, the dot product is returned as a Vector. If 'other' is an instance of Del, returns the directional derivative operator as a Python function. If this function is applied to a scalar expression, it returns the directional derivative of the scalar field wrt this Vector.

Parameters other: Vector/Dyadic/Del

The Vector or Dyadic we are dotting with, or a Del operator .

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> C.i.dot(C.j)
0
>>> C.i & C.i
1
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.dot(C.k)
5
>>> (C.i & delop)(C.x*C.y*C.z)
C.y*C.z
>>> d = C.i.outer(C.i)
>>> C.i.dot(d)
C.i
```

magnitude()

Returns the magnitude of this vector.

normalize()

Returns the normalized version of this vector.

outer(other)

Returns the outer product of this vector with another, in the form of a Dyadic instance.

Parameters other : Vector

The Vector with respect to which the outer product is to be computed.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
```

projection(other, scalar=False)

Returns the vector or scalar projection of the 'other' on 'self'.

Examples

```
>>> from sympy.vector.coordsysrect import CoordSys3D
>>> from sympy.vector.vector import Vector, BaseVector
>>> C = CoordSys3D('C')
>>> i, j, k = C.base_vectors()
>>> v1 = i + j + k
>>> v2 = 3*i + 4*j
>>> v1.projection(v2)
7/3*C.i + 7/3*C.j + 7/3*C.k
>>> v1.projection(v2, scalar=True)
7/3
```

separate()

The constituents of this vector in different coordinate systems, as per its definition. Returns a dict mapping each CoordSys3D to the corresponding constituent Vector.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> R1 = CoordSys3D('R1')
>>> R2 = CoordSys3D('R2')
>>> v = R1.i + R2.i
>>> v.separate() == {R1: R1.i, R2: R2.i}
True
```

to_matrix(system)

Returns the matrix form of this vector with respect to the specified coordinate system.

Parameters **system** : CoordSys3D

The system wrt which the matrix form is to be computed

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> from sympy.abc import a, b, c
>>> v = a*C.i + b*C.j + c*C.k
>>> v.to_matrix(C)
Matrix([
[a],
[b],
[c]])
```

Dyadic

class sympy.vector.dyadic.Dyadic

Super class for all Dyadic-classes.

References

[R560] (page 1873), [R561] (page 1873)

components

Returns the components of this dyadic in the form of a Python dictionary mapping BaseDyadic instances to the corresponding measure numbers.

cross(other)

Returns the cross product between this Dyadic, and a Vector, as a Vector instance.

Parameters other : Vector

The Vector that we are crossing this Dyadic with

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> d = N.i.outer(N.i)
>>> d.cross(N.j)
(N.i|N.k)
```

dot(other)

Returns the dot product(also called inner product) of this Dyadic, with another Dyadic or Vector. If 'other' is a Dyadic, this returns a Dyadic. Else, it returns a Vector (unless an error is encountered).

Parameters other : Dyadic/Vector

The other Dyadic or Vector to take the inner product with

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> D1 = N.i.outer(N.j)
>>> D2 = N.j.outer(N.j)
>>> D1.dot(D2)
(N.i|N.j)
>>> D1.dot(N.j)
N.i
```

to_matrix(system, second_system=None)

Returns the matrix form of the dyadic with respect to one or two coordinate systems.

Parameters system : CoordSys3D

The coordinate system that the rows and columns of the matrix correspond to. If a second system is provided, this only corresponds to the rows of the matrix.

second_system : CoordSys3D, optional, default=None

The coordinate system that the columns of the matrix correspond to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> v = N.i + 2*N.j
>>> d = v.outer(N.i)
>>> d.to_matrix(N)
Matrix([
[1, 0, 0],
[2, 0, 0],
[0, 0, 0]])
>>> from sympy import Symbol
>>> q = Symbol('q')
>>> P = N.orient_new_axis('P', q, N.k)
>>> d.to_matrix(N, P)
Matrix([
[ cos(q),  -sin(q),  0],
[2*cos(q), -2*sin(q), 0],
[      0,      0, 0]])
```

Del

class sympy.vector.deloperator.Del

Represents the vector differential operator, usually represented in mathematical expressions as the 'nabla' symbol.

cross(vect, doit=False)

Represents the cross product between this operator and a given vector - equal to the curl of the vector field.

Parameters **vect** : Vector

The vector whose curl is to be calculated.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> delop.cross(v, doit = True)
(-C.x*C.y + C.x*C.z)*C.i + (C.x*C.y - C.y*C.z)*C.j +
(-C.x*C.z + C.y*C.z)*C.k
>>> (delop ^ C.i).doit()
0
```

dot(vect, doit=False)

Represents the dot product between this operator and a given vector - equal to the divergence of the vector field.

Parameters **vect** : Vector

The vector whose divergence is to be calculated.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
>>> delop.dot(C.x*C.i)
Derivative(C.x, C.x)
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> (delop & v).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

gradient(scalar_field, doit=False)

Returns the gradient of the given scalar field, as a Vector instance.

Parameters **scalar_field** : SymPy expression

The scalar field to calculate the gradient of.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(9)
0
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Orienter classes (docstrings)

Orienter

class sympy.vector.orienters.**Orienter**

Super-class for all orienter classes.

rotation_matrix()

The rotation matrix corresponding to this orienter instance.

AxisOrienter

class sympy.vector.orienters.**AxisOrienter**(angle, axis)

Class to denote an axis orienter.

__init__(angle, axis)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

Parameters **angle** : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> from sympy.vector import AxisOrienter
>>> orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> B = N.orient_new('B', (orienter, ))
```

rotation_matrix(system)

The rotation matrix corresponding to this orienter instance.

Parameters **system** : CoordSys3D

The coordinate system wrt which the rotation matrix is to be computed

BodyOrienter

class sympy.vector.orienters.**BodyOrienter**(angle1, angle2, angle3, rot_order)

Class to denote a body-orienter.

__init__(angle1, angle2, angle3, rot_order)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see http://en.wikipedia.org/wiki/Euler_angles.

Parameters **angle1, angle2, angle3** : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```
>>> from sympy.vector import CoordSys3D, BodyOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> D = N.orient_new('D', (body_orienter, ))
```

is same as

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> D = N.orient_new('D', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, D.j)
>>> D = D.orient_new('D', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, D.k)
>>> D = D.orient_new('D', (axis_orienter3, ))
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> body_orienter1 = BodyOrienter(q1, q2, q3, '123')
>>> body_orienter2 = BodyOrienter(q1, q2, 0, 'ZXZ')
>>> body_orienter3 = BodyOrienter(0, 0, 0, 'YXX')
```

SpaceOrienter

class sympy.vector.orienters.**SpaceOrienter**(angle1, angle2, angle3, rot_order)

Class to denote a space-orienter.

__init__(angle1, angle2, angle3, rot_order)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters **angle1, angle2, angle3** : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

See also:

BodyOrienter (page 1799) Orienter to orient systems wrt Euler angles.

Examples

```
>>> from sympy.vector import CoordSys3D, SpaceOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> D = N.orient_new('D', (space_orienter, ))
```

is same as

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> B = N.orient_new('B', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, N.j)
>>> C = B.orient_new('C', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, N.k)
>>> D = C.orient_new('C', (axis_orienter3, ))
```

QuaternionOrienter

class sympy.vector.orienters.**QuaternionOrienter**(angle1, angle2, angle3, rot_order)

Class to denote a quaternion-orienter.

__init__(angle1, angle2, angle3, rot_order)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

$q_0 = \cos(\theta/2)$

$q_1 = \lambda_x \sin(\theta/2)$

$q_2 = \lambda_y \sin(\theta/2)$

$q_3 = \lambda_z \sin(\theta/2)$

Quaternion does not take in a rotation order.

Parameters **q0, q1, q2, q3** : Expr

The quaternions to rotate the coordinate system by

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> B = N.orient_new('B', (q_orienter, ))
```

Essential Functions in sympy.vector (docstrings)

matrix_to_vector

sympy.vector.**matrix_to_vector**(matrix, system)

Converts a vector in matrix form to a Vector instance.

It is assumed that the elements of the Matrix represent the measure numbers of the components of the vector along basis vectors of 'system'.

Parameters matrix : SymPy Matrix, Dimensions: (3, 1)

The matrix to be converted to a vector

system : CoordSys3D

The coordinate system the vector is to be defined in

Examples

```
>>> from sympy import ImmutableMatrix as Matrix
>>> m = Matrix([1, 2, 3])
>>> from sympy.vector import CoordSys3D, matrix_to_vector
>>> C = CoordSys3D('C')
>>> v = matrix_to_vector(m, C)
>>> v
C.i + 2*C.j + 3*C.k
>>> v.to_matrix(C) == m
True
```

express

`sympy.vector.express(expr, system, system2=None, variables=False)`

Global function for 'express' functionality.

Re-expresses a Vector, Dyadic or scalar(sympyifiable) in the given coordinate system.

If 'variables' is True, then the coordinate variables (base scalars) of other coordinate systems present in the vector/scalar field or dyadic are also substituted in terms of the base scalars of the given system.

Parameters expr : Vector/Dyadic/scalar(sympyifiable)

The expression to re-express in CoordSys3D 'system'

system: CoordSys3D

The coordinate system the expr is to be expressed in

system2: CoordSys3D

The other coordinate system required for re-expression (only for a Dyadic Expr)

variables : boolean

Specifies whether to substitute the coordinate variables present in expr, in terms of those of parameter system

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol, cos, sin
>>> N = CoordSys3D('N')
>>> q = Symbol('q')
```

(continues on next page)

(continued from previous page)

```

>>> B = N.orient_new_axis('B', q, N.k)
>>> from sympy.vector import express
>>> express(B.i, N)
(cos(q))*N.i + (sin(q))*N.j
>>> express(N.x, B, variables=True)
B.x*cos(q) - B.y*sin(q)
>>> d = N.i.outer(N.i)
>>> express(d, B, N) == (cos(q))*(B.i|N.i) + (-sin(q))*(B.j|N.i)
True

```

curl

`sympy.vector.curl(vect, coord_sys=None, doit=True)`

Returns the curl of a vector field computed wrt the base scalars of the given coordinate system.

Parameters `vect` : Vector

The vector operand

coord_sys : CoordSys3D

The coordinate system to calculate the gradient in. Deprecated since version 1.1

doit : bool

If True, the result is returned after calling `.doit()` on each component. Else, the returned expression contains Derivative instances

Examples

```

>>> from sympy.vector import CoordSys3D, curl
>>> R = CoordSys3D('R')
>>> v1 = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> curl(v1)
0
>>> v2 = R.x*R.y*R.z*R.i
>>> curl(v2)
R.x*R.y*R.j + (-R.x*R.z)*R.k

```

divergence

`sympy.vector.divergence(vect, coord_sys=None, doit=True)`

Returns the divergence of a vector field computed wrt the base scalars of the given coordinate system.

Parameters `vector` : Vector

The vector operand

coord_sys : CoordSys3D

The coordinate system to calculate the gradient in. Deprecated since version 1.1

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, divergence
>>> R = CoordSys3D('R')
>>> v1 = R.x*R.y*R.z * (R.i+R.j+R.k)
```

```
>>> divergence(v1)
R.x*R.y + R.x*R.z + R.y*R.z
>>> v2 = 2*R.y*R.z*R.j
>>> divergence(v2)
2*R.z
```

gradient

`sympy.vector.gradient(scalar_field, coord_sys=None, doit=True)`

Returns the vector gradient of a scalar field computed wrt the base scalars of the given coordinate system.

Parameters **scalar_field** : SymPy Expr

The scalar field to compute the gradient of

coord_sys : CoordSys3D

The coordinate system to calculate the gradient in Deprecated since version 1.1

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, gradient
>>> R = CoordSys3D('R')
>>> s1 = R.x*R.y*R.z
>>> gradient(s1)
R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> s2 = 5*R.x**2*R.z
>>> gradient(s2)
10*R.x*R.z*R.i + 5*R.x**2*R.k
```

is_conservative

`sympy.vector.is_conservative(field)`

Checks if a field is conservative.

Parameters **field** : Vector

The field to check for conservative property

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_conservative
>>> R = CoordSys3D('R')
>>> is_conservative(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_conservative(R.z*R.j)
False
```

is_solenoidal

`sympy.vector.is_solenoidal(field)`

Checks if a field is solenoidal.

Parameters **field** : Vector

The field to check for solenoidal property

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_solenoidal
>>> R = CoordSys3D('R')
>>> is_solenoidal(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_solenoidal(R.y * R.j)
False
```

scalar_potential

`sympy.vector.scalar_potential(field, coord_sys)`

Returns the scalar potential function of a field in a given coordinate system (without the added integration constant).

Parameters **field** : Vector

The vector field whose scalar potential function is to be calculated

coord_sys : CoordSys3D

The coordinate system to do the calculation in

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import scalar_potential, gradient
>>> R = CoordSys3D('R')
>>> scalar_potential(R.k, R) == R.z
```

(continues on next page)

(continued from previous page)

```
True
>>> scalar_field = 2*R.x**2*R.y*R.z
>>> grad_field = gradient(scalar_field)
>>> scalar_potential(grad_field, R)
2*R.x**2*R.y*R.z
```

scalar_potential_difference

`sympy.vector.scalar_potential_difference(field, coord_sys, point1, point2)`

Returns the scalar potential difference between two points in a certain coordinate system, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at point2) - (potential at point1)

The position vectors of the two Points are calculated wrt the origin of the coordinate system provided.

Parameters **field** : Vector/Expr

The field to calculate wrt

coord_sys : CoordSys3D

The coordinate system to do the calculations in

point1 : Point

The initial Point in given coordinate system

position2 : Point

The second Point in the given coordinate system

Examples

```
>>> from sympy.vector import CoordSys3D, Point
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', R.x*R.i + R.y*R.j + R.z*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
2*R.x**2*R.y
>>> Q = R.origin.locate_new('Q', 3*R.i + R.j + 2*R.k)
>>> scalar_potential_difference(vectfield, R, P, Q)
-2*R.x**2*R.y + 18
```

5.42.3 References for Vector

5.43 Contributions to docs

All contributions are welcome. If you'd like to improve something, look into the sources if they contain the information you need (if not, please fix them), otherwise the documentation generation needs to be improved (look in the `doc/` directory).

SYMPY SPECIAL TOPICS

6.1 Introduction

The purpose of this collection of documents is to provide users of SymPy with topics which are not strictly tutorial or are longer than tutorials and tests. The documents will hopefully fill a gap as SymPy matures and users find more ways to show how SymPy can be used in more advanced topics.

6.2 Finite Difference Approximations to Derivatives

6.2.1 Introduction

Finite difference approximations to derivatives is quite important in numerical analysis and in computational physics. In this tutorial we show how to use SymPy to compute approximations of varying accuracy. The hope is that these notes could be useful for the practicing researcher who is developing code in some language and needs to be able to efficiently generate finite difference formulae for various approximations.

In order to establish notation, we first state that we envision that there exists a continuous function F of a single variable x , with F having as many derivatives as desired. We sample x values uniformly at points along the real line separated by h . In most cases we want h to be small in some sense. $F(x)$ may be expanded about some point x_0 via the usual Taylor series expansion. Let $x = x_0 + h$. Then the Taylor expansion is

$$F(x_0 + h) = F(x_0) + \left(\frac{dF}{dx}\right)_{x_0} * h + \frac{1}{2!} \left(\frac{d^2F}{dx^2}\right)_{x_0} * h^2 + \frac{1}{3!} \left(\frac{d^3F}{dx^3}\right)_{x_0} * h^3 + \dots$$

In order to simplify the notation, we now define a set of coefficients c_n , where

$$c_n := \frac{1}{n!} \left(\frac{d^n F}{dx^n}\right)_{x_0}.$$

So now our series has the form:

$$F(x_0 + h) = F(x_0) + c_1 * h + c_2 * h^2 + c_3 * h^3 + \dots$$

In the following we will only use a finite grid of values x_i with i running from $1, \dots, N$ and the corresponding values of our function F at these grid points denoted by F_i . So the problem is how to generate approximate values for the derivatives of F with the constraint that we use a subset of the finite set of pairs (x_i, F_i) of size N .

What follows are manipulations using SymPy to formulate approximations for derivatives of a given order and to assess its accuracy. First, we use SymPy to derive the approximations

by using a rather brute force method frequently covered in introductory treatments. Later we shall make use of other SymPy functions which get the job done with more efficiency.

6.2.2 A Direct Method Using SymPy Matrices

If we let $x_0 = x_i$, evaluate the series at $x_{i+1} = x_i + h$ and truncate all terms above $O(h^1)$ we can solve for the single coefficient c_1 and obtain an approximation to the first derivative:

$$\left(\frac{dF}{dx}\right)_{x_0} \approx \frac{F_{i+1} - F_i}{h} + O(h)$$

where the $O(h)$ refers to the lowest order term in the series in h . This establishes that the derivative approximation is of first order accuracy. Put another way, if we decide that we can only use the two pairs (x_i, F_i) and (x_{i+1}, F_{i+1}) we obtain a “first order accurate” derivative.

In addition to (x_i, F_i) we next use the two points (x_{i+1}, F_{i+1}) and (x_{i+2}, F_{i+2}) . Then we have two equations:

$$F_{i+1} = F_i + c_1 * h + \frac{1}{2} * c_2 * h^2 + \frac{1}{3!} * c_3 * h^3 + \dots$$

$$F_{i+2} = F_i + c_1 * (2h) + \frac{1}{2} * c_2 * (2h)^2 + \frac{1}{3!} * c_3 * (2h)^3 + \dots$$

If we again want to find the first derivative (c_1), we can do that by eliminating the term involving c_2 from the two equations. We show how to do it using SymPy.

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_0, h')
>>> Fi, Fip1, Fip2 = symbols('F_{i}, F_{i+1}, F_{i+2}')
>>> n = 3 # there are the coefficients c_0=Fi, c_1=dF/dx, c_2=d**2F/dx**2
>>> c = symbols('c:3')
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
```

Vector of right hand sides:

```
>>> R = Matrix([[Fi], [Fip1], [Fip2]])
```

Now we make a matrix consisting of the coefficients of the c_i in the n th degree polynomial P . Coefficients of c_i evaluated at x_i :

```
>>> m11 = P(x0, x0, c, n).diff(c[0])
>>> m12 = P(x0, x0, c, n).diff(c[1])
>>> m13 = P(x0, x0, c, n).diff(c[2])
```

Coefficients of c_i evaluated at $x_i + h$:

```
>>> m21 = P(x0+h, x0, c, n).diff(c[0])
>>> m22 = P(x0+h, x0, c, n).diff(c[1])
>>> m23 = P(x0+h, x0, c, n).diff(c[2])
```

Coefficients of c_i evaluated at $x_i + 2 * h$:

```
>>> m31 = P(x0+2*h, x0, c, n).diff(c[0])
>>> m32 = P(x0+2*h, x0, c, n).diff(c[1])
>>> m33 = P(x0+2*h, x0, c, n).diff(c[2])
```


Matrix of the coefficients is 3x3 in this case:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Matrix form of the three equations for the c_i is $M \cdot X = R$:

The solution is obtained by directly inverting the 3x3 matrix M:

```
>>> X = M.inv() * R
```

Note that all three coefficients make up the solution. The desired first derivative is coefficient c_1 which is $X[1]$.

```
>>> print(together(X[1]))
(4*F_{i+1} - F_{i+2} - 3*F_{i})/(2*h)
```

It is instructive to compute another three-point approximation to the first derivative, except centering the approximation at x_i and thus using points at x_{i-1} , x_i , and x_{i+1} . So here is how this can be done using the 'brute force' method:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_i, h')
>>> Fi, Fim1, Fip1 = symbols('F_{i}, F_{i-1}, F_{i+1}')
>>> n = 3 # there are the coefficients c_0=Fi, c_1=dF/h, c_2=d**2F/h**2
>>> c = symbols('c:3')
>>> # define a polynomial of degree n
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
>>> # now we make a matrix consisting of the coefficients
>>> # of the c_i in the nth degree polynomial P
>>> # coefficients of c_i evaluated at x_i
>>> m11 = P(x0, x0, c, n).diff(c[0])
>>> m12 = P(x0, x0, c, n).diff(c[1])
>>> m13 = P(x0, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
>>> m21 = P(x0-h, x0, c, n).diff(c[0])
>>> m22 = P(x0-h, x0, c, n).diff(c[1])
>>> m23 = P(x0-h, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(x0+h, x0, c, n).diff(c[0])
>>> m32 = P(x0+h, x0, c, n).diff(c[1])
>>> m33 = P(x0+h, x0, c, n).diff(c[2])
>>> # matrix of the coefficients is 3x3 in this case
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Now that we have the matrix of coefficients we next form the right-hand-side and solve by inverting M :

```
>>> # matrix of the function values...actually a vector of right hand sides
>>> R = Matrix([[Fi], [Fim1], [Fip1]])
>>> # matrix form of the three equations for the c_i is M*X = R
>>> # solution directly inverting the 3x3 matrix M:
>>> X = M.inv() * R
>>> # note that all three coefficients make up the solution
>>> # the first derivative is coefficient c_1 which is X[1].
>>> print("The second-order accurate approximation for the first derivative is: ")
The second-order accurate approximation for the first derivative is:
```

(continues on next page)

(continued from previous page)

```
>>> print(together(X[1]))
(F_{i+1} - F_{i-1})/(2*h)
```

These two examples serve to show how one can directly find second order accurate first derivatives using SymPy. The first example uses values of x and F at all three points x_i , x_{i+1} , and x_{i+2} whereas the second example only uses values of x at the two points x_{i-1} and x_{i+1} and thus is a bit more efficient.

From these two simple examples a general rule is that if one wants a first derivative to be accurate to $O(h^n)$ then one needs $n+1$ function values in the approximating polynomial (here provided via the function $P(x, x_0, c, n)$).

Now let's assess the question of the accuracy of the centered difference result to see how we determine that it is really second order. To do this we take the result for dF/dx and substitute in the polynomial expansion for a higher order polynomial and see what we get. To this end, we make a set of eight coefficients d and use them to perform the check:

```
>>> d = symbols('c:8')
>>> dfdxcheck = (P(x0+h, x0, d, 8) - P(x0-h, x0, d, 8))/(2*h)
>>> print(simplify(dfdxcheck)) # so the appropriate cancellation of terms involving_
↪ 'h' happens
c1 + c3*h**2/6 + c5*h**4/120 + c7*h**6/5040
```

Thus we see that indeed the derivative is c_1 with the next term in the series of order h^2 .

However, it can quickly become rather tedious to generalize the direct method as presented above when attempting to generate a derivative approximation to high order, such as 6 or 8 although the method certainly works and using the present method is certainly less tedious than performing the calculations by hand.

As we have seen in the discussion above, the simple centered approximation for the first derivative only uses two point values of the (x_i, F_i) pairs. This works fine until one encounters the last point in the domain, say at $i = N$. Since our centered derivative approximation would use data at the point (x_{N+1}, F_{N+1}) we see that the derivative formula will not work. So, what to do? Well, a simple way to handle this is to devise a different formula for this last point which uses points for which we do have values. This is the so-called backward difference formula. To obtain it, we can use the same direct approach, except now use the three points (x_N, F_N) , (x_{N-1}, F_{N-1}) , and (x_{N-2}, F_{N-2}) and center the approximation at (x_N, F_N) . Here is how it can be done using SymPy:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, xN, h = symbols('x, x_N, h')
>>> FN, FNm1, FNm2 = symbols('F_{N}, F_{N-1}, F_{N-2}')
>>> n = 8 # there are the coefficients c_0=Fi, c_1=dF/h, c_2=d**2F/h**2
>>> c = symbols('c:8')
>>> # define a polynomial of degree d
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
```

Now we make a matrix consisting of the coefficients of the c_i in the d th degree polynomial P coefficients of c_i evaluated at x_i, x_{i-1} , and x_{i+1} :

```
>>> m11 = P(xN, xN, c, n).diff(c[0])
>>> m12 = P(xN, xN, c, n).diff(c[1])
>>> m13 = P(xN, xN, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
```

(continues on next page)

(continued from previous page)

```
>>> m21 = P(xN-h, xN, c, n).diff(c[0])
>>> m22 = P(xN-h, xN, c, n).diff(c[1])
>>> m23 = P(xN-h, xN, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(xN-2*h, xN, c, n).diff(c[0])
>>> m32 = P(xN-2*h, xN, c, n).diff(c[1])
>>> m33 = P(xN-2*h, xN, c, n).diff(c[2])
```

Next we construct the 3×3 matrix of the coefficients:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
>>> # matrix of the function values...actually a vector of right hand sides
>>> R = Matrix([[FN], [FNm1], [FNm2]])
```

Then we invert M and write the solution to the 3×3 system.

The matrix form of the three equations for the c_i is $M * C = R$. The solution is obtained by directly inverting M :

```
>>> X = M.inv() * R
```

The first derivative is coefficient c_1 which is $X[1]$. Thus the second order accurate approximation for the first derivative is:

```
>>> print("The first derivative centered at the last point on the right is:")
The first derivative centered at the last point on the right is:
>>> print(together(X[1]))
(-4*F_{N-1} + F_{N-2} + 3*F_{N})/(2*h)
```

Of course, we can devise a similar formula for the value of the derivative at the left end of the set of points at (x_1, F_1) in terms of values at (x_2, F_2) and (x_3, F_3) .

Also, we note that output of formats appropriate to Fortran, C, etc. may be done in the examples given above.

Next we show how to perform these and many other discretizations of derivatives, but using a much more efficient approach originally due to Bengt Fornberg and now incorporated into SymPy.

DEVELOPMENT TIPS: COMPARISONS IN PYTHON

7.1 Introduction

When debugging comparisons and hashes in SymPy, it is necessary to understand when exactly Python calls each method. Unfortunately, the official Python documentation for this is not very detailed (see the docs for [rich comparison](#), `__cmp__()` and `__hash__()` methods).

We wrote this guide to fill in the missing gaps. After reading it, you should be able to understand which methods do (and do not) get called and the order in which they are called.

7.2 Hashing

Every Python class has a `__hash__()` method, the default implementation of which is:

```
def __hash__(self):  
    return id(self)
```

You can reimplement it to return a different integer that you compute on your own. `hash(x)` just calls `x.__hash__()`. Python builtin classes usually redefine the `__hash__()` method. For example, an `int` has something like this:

```
def __hash__(self):  
    return int(self)
```

and a `list` does something like this:

```
def __hash__(self):  
    raise TypeError("list objects are unhashable")
```

The general idea about hashes is that if two objects have a different hash, they are not equal, but if they have the same hash, they might be equal. (This is usually called a “hash collision” and you need to use the methods described in the next section to determine if the objects really are equal).

The only requirement from the Python side is that the hash value mustn’t change after it is returned by the `__hash__()` method.

Please be aware that hashing is platform-dependent. This means that you can get different hashes for the same SymPy object on different platforms. This affects for instance sorting of sympy expressions. You can also get SymPy objects printed in different order.

When developing, you have to be careful about this, especially when writing tests. It is possible that your test runs on a 32-bit platform, but not on 64-bit. An example:

```
>> from sympy import *
>> x = Symbol('x')
>> r = rootfinding.roots_quartic(Poly(x**4 - 6*x**3 + 17*x**2 - 26*x + 20, x))
>> [i.evalf(2) for i in r]
[1.0 + 1.7*I, 2.0 - 1.0*I, 2.0 + I, 1.0 - 1.7*I]
```

If you get this order of solutions, you are probably running 32-bit system. On a 64-bit system you would get the following:

```
>> [i.evalf(2) for i in r]
[1.0 - 1.7*I, 1.0 + 1.7*I, 2.0 + I, 2.0 - 1.0*I]
```

When you now write a test like this:

```
r = [i.evalf(2) for i in r]
assert r == [1.0 + 1.7*I, 2.0 - 1.0*I, 2.0 + I, 1.0 - 1.7*I]
```

it will fail on a 64-bit platforms, even if it works for your 32-bit system. You can avoid this by using the `sorted()` or `set()` Python built-in:

```
r = [i.evalf(2) for i in r]
assert set(r) == set([1.0 + 1.7*I, 2.0 - 1.0*I, 2.0 + I, 1.0 - 1.7*I])
```

This approach does not work for doctests since they always compare strings that would be printed after a prompt. In that case you could make your test print results using a combination of `str()` and `sorted()`:

```
>> sorted([str(i.evalf(2)) for i in r])
['1.0 + 1.7*I', '1.0 - 1.7*I', '2.0 + I', '2.0 - 1.0*I']
```

or, if you don't want to show the values as strings, then sympify the results or the sorted list:

```
>> [S(s) for s in sorted([str(i.evalf(2)) for i in r])]
[1.0 + 1.7*I, 1.0 - 1.7*I, 2.0 + I, 2.0 - I]
```

The printing of SymPy expressions might be also affected, so be careful with doctests. If you get the following on a 32-bit system:

```
>> print dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) - f(x), f(x))
f(x) == C1*exp(-x + x*sqrt(2)) + C2*exp(-x - x*sqrt(2))
```

you might get the following on a 64-bit platform:

```
>> print dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) - f(x), f(x))
f(x) == C1*exp(-x - x*sqrt(2)) + C2*exp(-x + x*sqrt(2))
```

7.3 Method Resolution

Let `a`, `b` and `c` be instances of any one of the Python classes. As can be easily checked by the [Python script](#) (page 1818) at the end of this guide, if you write:

```
a == b
```

Python calls the following – in this order:

```
a.__eq__(b)
b.__eq__(a)
a.__cmp__(b)
b.__cmp__(a)
id(a) == id(b)
```

If a particular method is not implemented (or a method returns `NotImplemented`¹) Python skips it and tries the next one until it succeeds (i.e. until the method returns something meaningful). The last line is a catch-all method that always succeeds.

If you write:

```
a != b
```

Python tries to call:

```
a.__ne__(b)
b.__ne__(a)
a.__cmp__(b)
b.__cmp__(a)
id(a) == id(b)
```

If you write:

```
a < b
```

Python tries to call:

```
a.__lt__(b)
b.__gt__(a)
a.__cmp__(b)
b.__cmp__(a)
id(a) < id(b)
```

If you write:

```
a <= b
```

Python tries to call:

```
a.__le__(b)
b.__ge__(a)
a.__cmp__(b)
b.__cmp__(a)
id(a) <= id(b)
```

And similarly for `a > b` and `a >= b`.

If you write:

```
sorted([a, b, c])
```

Python calls the same chain of methods as for the `b < a` and `c < b` comparisons.

If you write any of the following:

¹ There is also the similar `NotImplementedError` exception, which one may be tempted to raise to obtain the same effect as returning `NotImplemented`.

But these are **not** the same, and Python will completely ignore `NotImplementedError` with respect to choosing appropriate comparison method, and will just propagate this exception upwards, to the caller.

So return `NotImplemented` is not the same as `raise NotImplementedError`.

```
a in {d: 5}
a in set([d, d, d])
set([a, b]) == set([a, b])
```

Python first compares hashes, e.g.:

```
a.__hash__()
d.__hash__()
```

If `hash(a) != hash(d)` then the result of the statement `a in {d: 5}` is immediately `False` (remember how hashes work in general). If `hash(a) == hash(d)` Python goes through the method resolution of the `==` operator as shown above.

7.4 General Notes and Caveats

In the method resolution for `<`, `<=`, `==`, `!=`, `>=`, `>` and `sorted([a, b, c])` operators the `__hash__()` method is not called, so in these cases it doesn't matter what it returns. The `__hash__()` method is only called for sets and dictionaries.

In the official Python documentation you can read about [hashable](#) and [non-hashable](#) objects. In reality, you don't have to think about it, you just follow the method resolution described here. E.g. if you try to use lists as dictionary keys, the list's `__hash__()` method will be called and it returns an exception.

In SymPy, every instance of any subclass of `Basic` is immutable. Technically this means, that its behavior through all the methods above mustn't change once the instance is created. Especially, the hash value mustn't change (as already stated above) or else objects will get mixed up in dictionaries and wrong values will be returned for a given key, etc....

7.5 Script To Verify This Guide

The above method resolution can be verified using the following program:

```
class A(object):

    def __init__(self, a, hash):
        self.a = a
        self._hash = hash

    def __lt__(self, o):
        print "%s.__lt__(%s)" % (self.a, o.a)
        return NotImplemented

    def __le__(self, o):
        print "%s.__le__(%s)" % (self.a, o.a)
        return NotImplemented

    def __gt__(self, o):
        print "%s.__gt__(%s)" % (self.a, o.a)
        return NotImplemented

    def __ge__(self, o):
```

(continues on next page)

(continued from previous page)

```

    print "%s.__ge__(%s)" % (self.a, o.a)
    return NotImplemented

    def __cmp__(self, o):
        print "%s.__cmp__(%s)" % (self.a, o.a)
        #return cmp(self._hash, o._hash)
        return NotImplemented

    def __eq__(self, o):
        print "%s.__eq__(%s)" % (self.a, o.a)
        return NotImplemented

    def __ne__(self, o):
        print "%s.__ne__(%s)" % (self.a, o.a)
        return NotImplemented

    def __hash__(self):
        print "%s.__hash__()" % (self.a)
        return self._hash

def show(s):
    print "--- %s " % s + "-"*40
    eval(s)

a = A("a", 1)
b = A("b", 2)
c = A("c", 3)
d = A("d", 1)

show("a == b")
show("a != b")
show("a < b")
show("a <= b")
show("a > b")
show("a >= b")
show("sorted([a, b, c])")
show("{d: 5}")
show("a in {d: 5}")
show("set([d, d, d])")
show("a in set([d, d, d])")
show("set([a, b])")

print "--- x = set([a, b]); y = set([a, b]); ---"
x = set([a, b])
y = set([a, b])
print "                x == y :"
x == y

print "--- x = set([a, b]); y = set([b, d]); ---"
x = set([a, b])
y = set([b, d])
print "                x == y :"
x == y

```

and its output:

```
-- a == b -----
a.__eq__(b)
b.__eq__(a)
a.__cmp__(b)
b.__cmp__(a)
-- a != b -----
a.__ne__(b)
b.__ne__(a)
a.__cmp__(b)
b.__cmp__(a)
-- a < b -----
a.__lt__(b)
b.__gt__(a)
a.__cmp__(b)
b.__cmp__(a)
-- a <= b -----
a.__le__(b)
b.__ge__(a)
a.__cmp__(b)
b.__cmp__(a)
-- a > b -----
a.__gt__(b)
b.__lt__(a)
a.__cmp__(b)
b.__cmp__(a)
-- a >= b -----
a.__ge__(b)
b.__le__(a)
a.__cmp__(b)
b.__cmp__(a)
-- sorted([a, b, c]) -----
b.__lt__(a)
a.__gt__(b)
b.__cmp__(a)
a.__cmp__(b)
c.__lt__(b)
b.__gt__(c)
c.__cmp__(b)
b.__cmp__(c)
-- {d: 5} -----
d.__hash__()
-- a in {d: 5} -----
d.__hash__()
a.__hash__()
d.__eq__(a)
a.__eq__(d)
d.__cmp__(a)
a.__cmp__(d)
-- set([d, d, d]) -----
d.__hash__()
d.__hash__()
d.__hash__()
-- a in set([d, d, d]) -----
d.__hash__()
d.__hash__()
d.__hash__()
a.__hash__()
```

(continues on next page)

(continued from previous page)

```
d.__eq__(a)
a.__eq__(d)
d.__cmp__(a)
a.__cmp__(d)
--- set([a, b]) -----
a.__hash__()
b.__hash__()
--- x = set([a, b]); y = set([a, b]); ---
a.__hash__()
b.__hash__()
a.__hash__()
b.__hash__()
                x == y :
--- x = set([a, b]); y = set([b, d]); ---
a.__hash__()
b.__hash__()
b.__hash__()
d.__hash__()
                x == y :
d.__eq__(a)
a.__eq__(d)
d.__cmp__(a)
a.__cmp__(d)
```


SymPy has a public wiki located at <http://wiki.sympy.org>. Users should feel free to contribute to this wiki anything interesting/useful.

8.1 FAQ

[FAQ](#) is one of the most useful wiki pages. It has answers to frequently-asked questions.

SYMPY PAPERS

A list of papers which either use or mention SymPy, as well as presentations given about SymPy at conferences can be seen at [SymPy Papers](#).

PLANET SYMPY

We have a blog aggregator at <http://planet.sympy.org>.

SYMPY LOGOS

SymPy has a collection of official logos, which can be generated from [sympy.svg](#) in your local copy of SymPy by:

```
$ cd doc  
$ make logo # will be stored in the _build/logo subdirectory
```

The license of all the logos is the same as SymPy: BSD. See the LICENSE file in the trunk for more information.

BLOGS, NEWS, MAGAZINES

You can see a list of blogs/news/magazines mentioning SymPy (that we know of) on our [wiki page](#).

13.1 SymPy Development Team

All people who contributed to SymPy by sending at least a patch or more (in the order of the date of their first contribution), except those who explicitly didn't want to be mentioned. People with a * next to their names are not found in the metadata of the git history. This file is generated automatically by running './bin/authors_update.py'.

There are a total of 739 authors.

Ondřej Čertík <ondrej@certik.cz>
Fabian Pedregosa <fabian@fseoane.net>
Jurjen N.E. Bos <jnebos@gmail.com>
Mateusz Paprocki <mattpap@gmail.com>
*Marc-Etienne M.Levéille <protonyc@gmail.com>
Brian Jorgensen <brian.jorgensen@gmail.com>
Jason Gedge <inferno1386@gmail.com>
Robert Schwarz <lethargo@googlemail.com>
Pearu Peterson <pearu.peterson@gmail.com>
Fredrik Johansson <fredrik.johansson@gmail.com>
Chris Wu <chris.wu@gmail.com>
*Ulrich Hecht <ulrich.hecht@gmail.com>
Goutham Lakshminarayan <dl.goutham@gmail.com>
David Lawrence <dmlawrence@gmail.com>
Jarosław Tworek <dev.jrx@gmail.com>
David Marek <h4wk.cz@gmail.com>
Bernhard R. Link <brlink@debian.org>
Andrej Tokarčík <androsis@gmail.com>
Or Dvory <gidesa@gmail.com>
Saroj Adhikari <adh.saroj@gmail.com>
Pauli Virtanen <pav@iki.fi>
Robert Kern <robert.kern@gmail.com>
James Aspnes <aspnes@cs.yale.edu>
Nimish Telang <ntelang@gmail.com>
Abderrahim Kitouni <a.kitouni@gmail.com>
Pan Peng <pengpanster@gmail.com>
Friedrich Hagedorn <friedrich_h@gmx.de>
Elrond der Elbenfuerst <elrond+sympy.org@samba-tng.org>
Rizgar Mella <rizgar.mella@gmail.com>
Felix Kaiser <felix.kaiser@fxkr.net>
Roberto Nobrega <rnobrega@gmail.com>
David Roberts <dvdr18@gmail.com>
Sebastian Krämer <basti.kr@gmail.com>

(continues on next page)

(continued from previous page)

Vinzent Steinberg <vinzent.steinberg@gmail.com>
Riccardo Gori <goriccardo@gmail.com>
Case Van Horsen <casevh@gmail.com>
Stepan Roucka <stepan@roucka.eu>
Ali Raza Syed <arsyed@gmail.com>
Stefano Maggiolo <s.maggiolo@gmail.com>
Robert Cimrman <cimrman3@ntc.zcu.cz>
Bastian Weber <bastian.weber@gmx-topmail.de>
Sebastian Krause <sebastian.krause@gmx.de>
Sebastian Kreft <skreft@gmail.com>
*Dan <coolg49964@gmail.com>
Alan Bromborsky <abrombo@verizon.net>
Boris Timokhin <qoqenator@gmail.com>
Robert <average.programmer@gmail.com>
Andy R. Terrel <aterrel@uchicago.edu>
Hubert Tsang <intsangity@gmail.com>
Konrad Meyer <konrad.meyer@gmail.com>
Henrik Johansson <henjo2006@gmail.com>
Priit Laes <plaes@plaes.org>
Freddie Witherden <freddie@witherden.org>
Brian E. Granger <ellisonbg@gmail.com>
Andrew Straw <strawman@astraw.com>
Kaifeng Zhu <cafeeee@gmail.com>
Ted Horst <ted.horst@earthlink.net>
Andrew Docherty <andrewd@maths.usyd.edu.au>
Akshay Srinivasan <akshaysrinivasan@gmail.com>
Aaron Meurer <asmeurer@gmail.com>
Barry Wardell <barry.wardell@gmail.com>
Tomasz Buchert <thinred@gmail.com>
Vinay Kumar <gnulinooks@gmail.com>
Johann Cohen-Tanugi <johann.cohentanugi@gmail.com>
Jochen Voss <voss@seehuhn.de>
Luke Peterson <hazelnusse@gmail.com>
Chris Smith <smichr@gmail.com>
Thomas Sidoti <TSidoti@gmail.com>
Florian Mickler <florian@mickler.org>
Nicolas Pourcelot <nicolas.pourcelot@gmail.com>
Ben Goodrich <goodrich.ben@gmail.com>
Toon Verstraelen <Toon.Verstraelen@UGent.be>
Ronan Lamy <ronan.lamy@gmail.com>
James Abbatiello <abbeyj@gmail.com>
Ryan Krauss <ryanlists@gmail.com>
Bill Flynn <wflynn@gmail.com>
Kevin Goodsell <kevin-opensource@omegacrash.net>
Jorn Baayen <jorn.baayen@gmail.com>
Eh Tan <tan2tan2@gmail.com>
Renato Coutinho <renato.coutinho@gmail.com>
Oscar Benjamin <oscar.j.benjamin@gmail.com>
Øyvind Jensen <jensen.oyvind@gmail.com>
Julio Idichekop Filho <idichekop@yahoo.com.br>
Łukasz Pankowski <lukpank@o2.pl>
*Chu-Ching Huang <cchuang@mail.cgu.edu.tw>
Fernando Perez <Fernando.Perez@berkeley.edu>
Raffaele De Feo <alberthilbert@gmail.com>
Christian MuiSe <christian.muise@gmail.com>
Matt Curry <mattjcurry@gmail.com>

(continues on next page)

(continued from previous page)

Kazuo Thow <kazuo.thow@gmail.com>
 Christian Schubert <chr.schubert@gmx.de>
 Jezreel Ng <jezreel@gmail.com>
 James Pearson <xiong.chiamiov@gmail.com>
 Matthew Brett <matthew.brett@gmail.com>
 Addison Cugini <ajcugini@gmail.com>
 Nicholas J.S. Kinar <n.kinar@usask.ca>
 Harold Erbin <harold.erbin@gmail.com>
 Thomas Dixon <thom@thomdixon.org>
 Cristóvão Sousa <crisjss@gmail.com>
 Andre de Fortier Smit <freevryheid@gmail.com>
 Mark Dewing <markdewing@gmail.com>
 Alexey U. Gudchenko <proga@goodok.ru>
 Gary Kerr <gary.kerr@blueyonder.co.uk>
 Sherjil Ozair <sherjilozair@gmail.com>
 Oleksandr Gituliar <gituliar@gmail.com>
 Sean Vig <sean.v.775@gmail.com>
 Prafullkumar P. Tale <hector1618@gmail.com>
 Vladimir Perić <vlada.peric@gmail.com>
 Tom Bachmann <e_mc_h2@web.de>
 Yuri Karadzhov <yuri.karadzhov@gmail.com>
 Vladimir Lagunov <werehuman@gmail.com>
 Matthew Rocklin <mrocklin@cs.uchicago.edu>
 Saptarshi Mandal <sapta.iitkgp@gmail.com>
 Gilbert Gede <gilbertgede@gmail.com>
 Anatolii Koval <weralwolf@gmail.com>
 Tomo Lazovich <lazovich@gmail.com>
 Pavel Fedotov <fedotovp@gmail.com>
 Jack McCaffery <jpmccaffery@gmail.com>
 Jeremias Yehdeghe <j.yehdeghe@gmail.com>
 Kibeom Kim <kk1674@nyu.edu>
 Gregory Ksionda <ksiondag846@gmail.com>
 Tomáš Bambas <tomas.bambas@gmail.com>
 Raymond Wong <rayman_407@yahoo.com>
 Luca Weihs <astronomicalcuriosity@gmail.com>
 Shai 'Deshe' Wyborski <shaide@cs.huji.ac.il>
 Thomas Wiecki <thomas.wiecki@gmail.com>
 Óscar Nájera <najera.oscar@gmail.com>
 Mario Pernici <mario.pernici@gmail.com>
 Benjamin McDonald <mcdonald.ben@gmail.com>
 Sam Magura <samtheman132@gmail.com>
 Stefan Krastanov <krastanov.stefan@gmail.com>
 Bradley Froehle <brad.froehle@gmail.com>
 Min Ragan-Kelley <benjaminrk@gmail.com>
 Emma Hogan <ehogan@gemini.edu>
 Nikhil Sarda <diff.operator@gmail.com>
 Julien Rioux <julien.rioux@gmail.com>
 Roberto Colistete, Jr. <roberto.colistete@gmail.com>
 Raoul Bourquin <raoulb@bluewin.ch>
 Gert-Ludwig Ingold <gert.ingold@physik.uni-augsburg.de>
 Srinivas Vasudevan <srvasude@gmail.com>
 Jason Moore <moorepants@gmail.com>
 Miha Marolt <tloramus@gmail.com>
 Tim Lahey <tim.lahey@gmail.com>
 Luis Garcia <ppn.online@me.com>
 Matt Rajca <matt.rajca@me.com>

(continues on next page)

(continued from previous page)

David Li <l33tnerd.li@gmail.com>
Alexandr Gudulin <alexandr.gudulin@gmail.com>
Bilal Akhtar <bilalakhtar@ubuntu.com>
Grzegorz Świrski <sognat@gmail.com>
Matt Habel <habelinc@gmail.com>
David Ju <Sgtmook314@gmail.com>
Nichita Utii <nikita.utii+github@gmail.com>
Nikolay Lazarov <qwerqwerqwer@abv.bg>
Steve Anton <anxui.nx@gmail.com>
Imran Ahmed Manzoor <imran.manzoor31@gmail.com>
Ljubiša Moćić <3rdslasher@gmail.com>
Piotr Korgul <p.korgul@gmail.com>
Jim Zhang <Hyriodula@gmail.com>
Sam Sleight <samuel.sleight@gmail.com>
tborisova <ts.borisova3@gmail.com>
Chancellor Arkantos <Chancellor_Arkantos@hotmail.co.uk>
Stepan Simsa <simsa.st@gmail.com>
Tobias Lenz <t_lenz94@web.de>
Siddhanathan Shanmugam <siddhanathan@gmail.com>
Tiffany Zhu <bubble.wubble.303@gmail.com>
Tristan Hume <tris.hume@gmail.com>
Alexey Subach <alexey.subach@gmail.com>
Joan Creus <joan.creus.c@gmail.com>
Geoffry Song <goffrie@gmail.com>
Puneeth Chaganti <punchagan@gmail.com>
Marcin Kostrzewa <>
Natalia Nawara <fankalemura@gmail.com>
vishal <vishal.panjwani15@gmail.com>
Shruti Mangipudi <shruti2395@gmail.com>
Davy Mao <e_equals_mass_speed_light_squared@hotmail.com>
Swapnil Agarwal <swapnilag29@gmail.com>
Dhia Kennouche <kendhia@gmail.com>
jerrymall121 <jerrymall121@gmail.com>
Joachim Durchholz <jo@durchholz.org>
Martin Povišer <martin.povik@gmail.com>
Siddhant Jain <getsiddhant@gmail.com>
Kevin Hunter <hunteke@earlham.edu>
Michael Mayorov <marchael@kb.csu.ru>
Nathan Alison <nathan.f.alison@gmail.com>
Christian Bühler <christian@cbuehler.de>
Carsten Knoll <CarstenKnoll@gmx.de>
Bharath M R <catchmrbharath@gmail.com>
Matthias Toews <mat.toews@googlemail.com>
Sergiu Ivanov <unlimitedscolobb@gmail.com>
Jorge E. Cardona <jorgeecardona@gmail.com>
Sanket Agarwal <sanket@sanketagarwal.com>
Manoj Babu K. <manoj.babu2378@gmail.com>
Sai Nikhil <tsnlegend@gmail.com>
Aleksandar Makelov <amakelov@college.harvard.edu>
Sachin Irukula <sachin.irukula@gmail.com>
Raphael Michel <webmaster@raphaelmichel.de>
Ashwini Oruganti <ashwini.oruganti@gmail.com>
Andreas Kloeckner <inform@tiker.net>
Prateek Papriwal <papriwalprateek@gmail.com>
Arpit Goyal <agmps18@gmail.com>
Angadh Nanjangud <angadh.n@gmail.com>

(continues on next page)

(continued from previous page)

Comer Duncan <comer.duncan@gmail.com>
 Jens H. Nielsen <jenshnielsen@gmail.com>
 Joseph Dougherty <Github@JWDougherty.com>
 Elliot Marshall <Marshall2389@gmail.com>
 Guru Devanla <grdvn@gmail.com>
 George Waksman <waksman@gwax.com>
 Alexandr Popov <alexandr.s.popov@gmail.com>
 Tarun Gaba <tarun.gaba7@gmail.com>
 Takafumi Arakaki <aka.tkf@gmail.com>
 Saurabh Jha <saurabh.jhaa@gmail.com>
 Rom le Clair <jacen.guardian@gmail.com>
 Angus Griffith <l6sn6uv@gmail.com>
 Timothy Reluga <treluga@math.psu.edu>
 Brian Stephanik <xoedusk@gmail.com>
 Alexander Eberspächer <alex.eberspaecher@gmail.com>
 Sachin Joglekar <srjoglekar246@gmail.com>
 Tyler Pirtle <teeler@gmail.com>
 Vasily Povalyaev <vapovalyaev@gmail.com>
 Colleen Lee <colleenclee@gmail.com>
 Matthew Hoff <mhoff14@gmail.com>
 Niklas Thörne <notrupertthorne@gmail.com>
 Huijun Mai <m.maihuijun@gmail.com>
 Marek Šuppa <mr@shu.io>
 Ramana Venkata <idlike2dream@gmail.com>
 Prasoon Shukla <prasoon92.iitr@gmail.com>
 Stefen Yin <zqyin@ucdavis.edu>
 Thomas Hisch <t.hisch@gmail.com>
 Madeleine Ball <mpball@gmail.com>
 Mary Clark <mary.spriteling@gmail.com>
 Rishabh Dixit <rishabhdixit11@gmail.com>
 Manoj Kumar <manojkumarsivaraj334@gmail.com>
 Akshit Agarwal <akshit.jiit@gmail.com>
 CJ Carey <perimosocordiae@gmail.com>
 Patrick Lacasse <patrick.m.lacasse@gmail.com>
 Ananya H <ananyaha93@gmail.com>
 Tarang Patel <tarangrockr@gmail.com>
 Christopher Dembia <cld72@cornell.edu>
 Benjamin Fishbein <fishbeinb@gmail.com>
 Sean Ge <seange727@gmail.com>
 Amit Jamadagni <bitsjamadagni@gmail.com>
 Ankit Agrawal <aaaagrawal@iitb.ac.in>
 Björn Dahlgren <bjodah@gmail.com>
 Christophe Saint-Jean <christophe.saint-jean@univ-lr.fr>
 Demian Wassermann <demian@bwh.harvard.edu>
 Khagesh Patel <khageshpatel93@gmail.com>
 Stephen Loo <shikil@yahoo.com>
 hm <hacman0@gmail.com>
 Patrick Poitras <acebulf@gmail.com>
 Katja Sophie Hotz <katja.sophie.hotz@student.tuwien.ac.at>
 Varun Joshi <joshi.142@osu.edu>
 Chetna Gupta <cheta.gup@gmail.com>
 Thilina Rathnayake <thilinarmtb@gmail.com>
 Max Hutchinson <maxhutch@gmail.com>
 Shravas K Rao <shravas@gmail.com>
 Matthew Tadd <matt.tadd@gmail.com>
 Alexander Hirzel <alex@hirzel.us>

(continues on next page)

(continued from previous page)

Randy Heydon <randy.heydon@clockworklab.net>
Oliver Lee <oliverzlee@gmail.com>
Seshagiri Prabhu <seshagiriprabhu@gmail.com>
Pradyumna <pradyul1993@gmail.com>
Erik Welch <erik.n.welch@gmail.com>
Eric Nelson <eric.the.red.XLII@gmail.com>
Roland Puntaier <roland.puntaier@chello.at>
Chris Conley <chrisconley15@gmail.com>
Tim Swast <tswast@gmail.com>
Dmitry Batkovich <batya239@gmail.com>
Francesco Bonazzi <franz.bonazzi@gmail.com>
Yuriy Demidov <iurii.demidov@gmail.com>
Rick Muller <rpmuller@gmail.com>
Manish Gill <gill.manish90@gmail.com>
Markus Müller <markus.mueller.1.g@googlemail.com>
Amit Saha <amitsaha.in@gmail.com>
Jeremy <twobitlogic@gmail.com>
QuaBoo <kisonchristian@gmail.com>
Stefan van der Walt <stefan@sun.ac.za>
David Joyner <wdjoyner@gmail.com>
Lars Buitinck <larsmans@gmail.com>
Alkiviadis G. Akritas <akritas@uth.gr>
Vinit Ravishankar <vinit.ravishankar@gmail.com>
Mike Boyle <boyle@astro.cornell.edu>
Heiner Kirchhoffer <Heiner.Kirchhoffer@gmail.com>
Pablo Puente <ppuedom@gmail.com>
James Fiedler <jrfiedler@gmail.com>
Harsh Gupta <mail@hargup.in>
Tuomas Airaksinen <tuomas.airaksinen@gmail.com>
Paul Strickland <p.e.strickland@gmail.com>
James Goppert <james.goppert@gmail.com>
rathmann <rathmann.os@gmail.com>
Avichal Dayal <avichal.dayal@gmail.com>
Paul Scott <paul.scott@nicta.com.au>
Shipra Banga <bangashipra@gmail.com>
Pramod Ch <pramodch14@gmail.com>
Akshay <akshaynukala95@gmail.com>
Buck Shlegeris <buck2@bruceh15.anu.edu.au>
Jonathan Miller <jdmiller93@gmail.com>
Edward Schembor <eschemb1@jhu.edu>
Rajath Shashidhara <rajaths.rajaths@gmail.com>
Zamrath Nizam <zamiguy_ni@yahoo.com>
Aditya Shah <adityashah30@gmail.com>
Rajat Aggarwal <rajataggarwal1975@gmail.com>
Sambuddha Basu <sammygamer@live.com>
Zeel Shah <kshah215@gmail.com>
Abhinav Chanda <abhinavchanda01@gmail.com>
Jim Crist <crist042@umn.edu>
Sudhanshu Mishra <mrsud94@gmail.com>
Anurag Sharma <anurags92@gmail.com>
Soumya Dipta Biswas <sdb1323@gmail.com>
Sushant Hiray <hiraysushant@gmail.com>
Ben Lucato <ben.lucato@gmail.com>
Kunal Arora <kunalarora.135@gmail.com>
Henry Gebhardt <hsggebhardt@gmail.com>
Dammina Sahabandu <dmsahabandu@gmail.com>

(continues on next page)

(continued from previous page)

Manish Shukla <manish.shukla393@gmail>
 Ralph Bean <rbean@redhat.com>
 richierichrawr <richierichrawr@users.noreply.github.com>
 John Connor <john.theman.connor@gmail.com>
 Juan Luis Cano Rodríguez <juanlu001@gmail.com>
 Sahil Shekhawat <sahilshekhawat01@gmail.com>
 Kundan Kumar <kundankumar18581@gmail.com>
 Stas Kelvich <stanconn@gmail.com>
 sevaader <sevaader@gmail.com>
 Dhruvesh Vijay Parikh <parikhdhruvesh1@gmail.com>
 Venkatesh Halli <venkatesh.fatality@gmail.com>
 Lennart Fricke <lennart@die-frickes.eu>
 Vlad Seghete <vlad.seghete@gmail.com>
 Shashank Agarwal <shashank.agarwal94@gmail.com>
 carstimon <carstimon@gmail.com>
 Pierre Haessig <pierre.haessig@crans.org>
 Maciej Baranski <getrox.sc@gmail.com>
 Benjamin Gudehus <hastebrot@gmail.com>
 Faisal Anees <faisal.iiit@gmail.com>
 Mark Shoulson <mark@kli.org>
 Robert Johansson <jrjohansson@gmail.com>
 Kalevi Suominen <jksuom@gmail.com>
 Kaushik Varanasi <kaushik.varanasi@gmail.com>
 Fawaz Alazemi <Mba7eth@gmail.com>
 Ambar Mehrotra <mehrotraambar@gmail.com>
 David P. Sanders <dpsanders@gmail.com>
 Peter Brady <petertbrady@gmail.com>
 John V. Siratt <jvsiratt@gmail.com>
 Sarwar Chahal <chahal.sarwar98@gmail.com>
 Nathan Woods <charlesnwoods@gmail.com>
 Colin B. Macdonald <cbm@m.fsf.org>
 Marcus Näslund <naslundx@gmail.com>
 Clemens Novak <clemens@familie-novak.net>
 Mridul Seth <seth.mridul@gmail.com>
 Craig A. Stoudt <craig.stoudt@gmail.com>
 Raj <raj454raj@gmail.com>
 Mihai A. Ionescu <ionescu.a.mihai@gmail.com>
 immerrrr <immerrrr@gmail.com>
 Chai Wah Wu <cwwuieeee@gmail.com>
 Leonid Blouvshtein <leonidbl91@gmail.com>
 Peleg Michaeli <freepeleg@gmail.com>
 ck Lux <lux.r.ck@gmail.com>
 zsc347 <zsc347@gmail.com>
 Hamish Dickson <hamish.dickson@gmail.com>
 Michael Gallaspy <gallaspy.michael@gmail.com>
 Roman Infiantskas <infroma@gmail.com>
 Duane Nykamp <nykamp@umn.edu>
 Ted Dokos <tdokos@gmail.com>
 Sunny Aggarwal <sunnyaggarwal1994@gmail.com>
 Victor Brebenar <v.brebenar@gmail.com>
 Akshat Jain <akshat.jain@students.iiit.ac.in>
 Shivam Vats <shivamvats.iitkgp@gmail.com>
 Longqi Wang <iqgnol@gmail.com>
 Juan Felipe Osorio <jfosorio@gmail.com>
 Ray Cathcart <github@cathcart.us>
 Lukas Zorich <lukas.zorich@gmail.com>

(continues on next page)

(continued from previous page)

Eric Miller <emiller42@gmail.com>
Cody Herbst <cyherbst@gmail.com>
Nishith Shah <nishithshah.2211@gmail.com>
Amit Kumar <dtu.amit@gmail.com>
Yury G. Kudryashov <urkud.urkud@gmail.com>
Guillaume Gay <contact@damcb.com>
Mihir Wadwekar <m.mihirw@gmail.com>
Tuan Manh Lai <laituan245@gmail.com>
Asish Panda <asishrocks95@gmail.com>
Darshan Chaudhary <deathbullet@gmail.com>
Alec Kalinin <alec.kalinin@gmail.com>
Ralf Stephan <ralf@ark.in-berlin.de>
Aaditya Nair <aadityanair6494@gmail.com>
Jayesh Lahori <jlahori92@gmail.com>
Harshil Goel <harshil158@gmail.com>
Luv Agarwal <agarwal.iiiit@gmail.com>
Jason Ly <jason.ly@gmail.com>
Lokesh Sharma <lokeshhsharma@gmail.com>
Sartaj Singh <singhsartaj94@gmail.com>
Chris Swierczewski <cswiercz@gmail.com>
Konstantin Togoi <konstantin.togoi@gmail.com>
Param Singh <paramsingh258@gmail.com>
Sumith Kulal <sumith1896@gmail.com>
Juha Remes <jremes@outlook.com>
Philippe Bouafia <philippe.bouafia@ensea.fr>
Peter Schmidt <peter@peterjs.com>
Jiaxing Liang <liangjiaxing57@gmail.com>
Lucas Jones <lucas@lucasjones.co.uk>
Gregory Ashton <gash789@gmail.com>
Jennifer White <jcrw122@googlemail.com>
Renato Orsino <renato.orsino@gmail.com>
Michael Boyle <michael.oliver.boyle@gmail.com>
Alistair Lynn <arplynn@gmail.com>
Govind Sahai <gsiitbhu@gmail.com>
Adam Bloomston <adam@glitterfram.es>
Kyle McDaniel <mcdanie5@illinois.edu>
Nguyen Truong Duy <truongduy134@yahoo.com>
Alex Lindsay <adlinds3@ncsu.edu>
Mathew Chong <mathewchong.dev@gmail.com>
Jason Siefken <siefkenj@gmail.com>
Gaurav Dhingra <gauravdhingra.gxyd@gmail.com>
Gao, Xiang <qasdfgtyuiop@gmail.com>
Kevin Ventullo <kevin.ventullo@gmail.com>
mao8 <thisisma08@gmail.com>
Isuru Fernando <isuruf@gmail.com>
Shivam Tyagi <shivam.tyagi.apml3@itbhu.ac.in>
Richard Otis <richard.otis@outlook.com>
Rich LaSota <rjlasota@gmail.com>
dustyrockpyle <dustyrockpyle@gmail.com>
Anton Akhmerov <anton.akhmerov@gmail.com>
Michael Zingale <michael.zingale@stonybrook.edu>
Chak-Pong Chung <chakpongchung@gmail.com>
David T <derDavidT@users.noreply.github.com>
Phil Ruffwind <rf@ruffwind.com>
Sebastian Koslowski <koslowski@kit.edu>
Kumar Krishna Agrawal <kumar.1994.14@gmail.com>

(continues on next page)

(continued from previous page)

Dustin Gadal <Dustin.Gadal@gmail.com>
 João Moura <operte@gmail.com>
 Yu Kobayashi <yukoba@accelart.jp>
 Shashank Kumar <shashank.kumar.apcl3@iitbhu.ac.in>
 Timothy Cyrus <tcyrus@users.noreply.github.com>
 Devyani Kota <devyanikota@gmail.com>
 Keval Shah <kevalshah_96@yahoo.co.in>
 Dzhelil Rufat <drufat@caltech.edu>
 Pastafarianist <mr.pastafarianist@gmail.com>
 Sourav Singh <souravsingh@users.noreply.github.com>
 Jacob Garber <jgarber1@ualberta.ca>
 Vinay Singh <csvinay.d@gmail.com>
 GolimarOurHero <metalera94@hotmail.com>
 Prashant Tyagi <prashanttyagi221295@gmail.com>
 Matthew Davis <davisml.md@gmail.com>
 Tschijnmo TSCHAU <tschijnmotschau@gmail.com>
 Alexander Bentkamp <bentkamp@gmail.com>
 Jack Kemp <metaknightdrake-git@yahoo.co.uk>
 Kshitij Saraogi <KshitijSaraogi@gmail.com>
 Thomas Baruchel <baruchel@gmx.com>
 Nicolás Guarín-Zapata <nicoguarin@gmail.com>
 Jens Jørgen Mortensen <jj@smoerhul.dk>
 Sampad Kumar Saha <sampadsaha5@gmail.com>
 Eva Charlotte Mayer <eva-charlotte.mayer@posteo.de>
 Laura Domine <temigo@gmx.com>
 Justin Blythe <jblythe29@gmail.com>
 Meghana Madhyastha <meghana.madhyastha@gmail.com>
 Tanu Hari Dixit <tokencolour@gmail.com>
 Shekhar Prasad Rajak <shekharrajak@live.com>
 Aqnouch Mohammed <aqnouch.mohammed@gmail.com>
 Arafat Dad Khan <arafat.da.khan@gmail.com>
 Boris Atamanovskiy <shaomoron@gmail.com>
 Sam Tygier <sam.tygier@hep.manchester.ac.uk>
 Jai Luthra <me@jailuthra.in>
 Guo Xingjian <Seeker1995@gmail.com>
 Sandeep Veethu <sandeep.veethu@gmail.com>
 Archit Verma <architv07@gmail.com>
 Shubham Tibra <shubh.tibra@gmail.com>
 Ashutosh Saboo <ashutosh.saboo96@gmail.com>
 Michael S. Hansen <michael.hansen@nih.gov>
 Anish Shah <shah.anish07@gmail.com>
 Guillaume Jacquenot <guillaume.jacquenot@gmail.com>
 Bhautik Mavani <mavanibhautik@gmail.com>
 Michał Radwański <enedil.isildur@gmail.com>
 Jerry Li <jerry@jerryli.ca>
 Pablo Zubieta <pabloferz@yahoo.com.mx>
 Shivam Agarwal <knowthyself2503@gmail.com>
 Chaitanya Sai Alaparthi <achaitanyasai@gmail.com>
 Arihant Parsoya <parsoyaarihant@gmail.com>
 Ruslan Pisarev <rpisarev@cloudlinux.com>
 Akash Trehan <akash.trehan123@gmail.com>
 Nishant Nikhil <nishantiam@gmail.com>
 Vladimir Poluhsin <vovapolu@gmail.com>
 Akshay Nagar <awesomeay13@yahoo.com>
 James Brandon Milam <jmilam343@gmail.com>
 Abhinav Agarwal <abhinavagarwal1996@gmail.com>

(continues on next page)

(continued from previous page)

Rishabh Daal <rishabhdaal@gmail.com>
Sanya Khurana <sanya@monica.in>
Aman Deep <amandeep1024@gmail.com>
Aravind Reddy <aravindreddy255@gmail.com>
Abhishek Verma <iamvermaabhishek@gmail.com>
Matthew Parnell <matt@parnmatt.co.uk>
Thomas Hickman <Thomas.Hickman42@gmail.com>
Akshay Siramdas <akshaysiramdas@gmail.com>
YiDing Jiang <yidinggjiaangg@gmail.com>
Jatin Yadav <jatinyadav25@gmail.com>
Matthew Thomas <mnmt@users.noreply.github.com>
Rehas Sachdeva <aquannie@gmail.com>
Michael Mueller <michaeldmueller7@gmail.com>
Srajan Garg <srajan.garg@gmail.com>
Prabhjot Singh <prabhjot.nith@gmail.com>
Haruki Moriguchi <harukimoriguchi@gmail.com>
Tom Gijselinck <tomgijselinck@gmail.com>
Nitin Chaudhary <nitinmax1000@gmail.com>
Alex Argunov <sajkoooo@gmail.com>
Nathan Musoke <nathan.musoke@gmail.com>
Abhishek Garg <abhishekgarg119@gmail.com>
Dana Jacobsen <dana@acm.org>
Vasiliy Dommès <vasdommes@gmail.com>
Phillip Berndt <phillip.berndt@googlemail.com>
Haimo Zhang <zh.hammer.dev@gmail.com>
Anthony Scopatz <scopatz@gmail.com>
bluebrook <perl4logic@gmail.com>
Leonid Kovalev <leonidvkoalev@gmail.com>
Josh Burkart <jburkart@gmail.com>
Dimitra Konomi <t8130064@dias.aueb.gr>
Christina Zografou <t8130048@dias.aueb.gr>
Fiach Antaw <fiach.antaw+github@gmail.com>
Langston Barrett <langston.barrett@gmail.com>
Krit Karan <kritkaran.b13@iiits.in>
G. D. McBain <gdmcbain@protonmail.com>
Prempal Singh <prempal.42@gmail.com>
Gabriel Orisaka <orisaka@gmail.com>
Matthias Bussonnier <bussonniermatthias@gmail.com>
rahuldan <rahul02013@gmail.com>
Colin Marquardt <github@marquardt-home.de>
Andrew Taber <andrew.e.taber@gmail.com>
Yash Reddy <write2yashreddy@gmail.com>
Peter Stangl <peter.stangl@ph.tum.de>
elvis-sik <e.sikora@grad.ufsc.br>
Nikos Karagiannakis <nikoskaragiannakis@gmail.com>
Jainul Vaghasia <jainulvaghasia@gmail.com>
Dennis Meckel <meckel@datenschuppen.de>
Harshil Meena <harshil.7535@gmail.com>
Micky <mickydroch@gmail.com>
Nick Curtis <nicholas.curtis@uconn.edu>
Michele Zaffalon <michele.zaffalon@gmail.com>
Martha Giannoudovardi <maapxa@gmail.com>
Devang Kulshreshtha <devang.kulshreshtha.cse14@itbhu.ac.in>
Steph Papanik <spapanik21@gmail.com>
Mohammad Sadeq Dousti <msdousti@gmail.com>
Arif Ahmed <arif.ahmed.5.10.1995@gmail.com>

(continues on next page)

(continued from previous page)

Abdullah Javed Nesar <abduljaved1994@gmail.com>
 Lakshya Agrawal <zeeshan.lakshya@gmail.com>
 shruti <shrutishrm512@gmail.com>
 Rohit Rango <rohit.rango@gmail.com>
 Hong Xu <hong@topbug.net>
 Ivan Petuhov <ivan@ostrovok.ru>
 Alsheh <alsheh@rpi.edu>
 Marcel Stimberg <marcel.stimberg@ens.fr>
 Alexey Pakhokmhik <cool.Bakov@yandex.ru>
 Tommy Olofsson <tommy.olofsson.90@gmail.com>
 Zulfikar <zulfikar97@gmail.com>
 Blair Azzopardi <bsdz@users.noreply.github.com>
 Danny Hermes <daniel.j.hermes@gmail.com>
 Sergey Pestov <pestov-sa@yandex.ru>
 Mohit Chandra <mohit.chandra@research.iiit.ac.in>
 Karthik Chintapalli <karthik.chintapalli@students.iiit.ac.in>
 Marcin Briański <marcin.brianski@student.uj.edu.pl>
 andreo <andrey.torba@gmail.com>
 Flamy Owl <flamyowl@protonmail.ch>
 Yicong Guo <guoyicong100@gmail.com>
 Varun Garg <varun.garg03@gmail.com>
 Rishabh Madan <rishabhmadan96@gmail.com>
 Aditya Kapoor <aditya.kapoor.apml2@itbhu.ac.in>
 Karan Sharma <karan1276@gmail.com>
 Vedant Rathore <vedantr1998@gmail.com>
 Johan Blåbäck <johan_bluecreek@riseup.net>
 Pranjal Tale <pranjaltale16@gmail.com>
 Jason Tokayer <jason.tokayer@gmail.com>
 Raghav Jajodia <jajodia.rahav@gmail.com>
 Rajat Thakur <rajatthakur1997@gmail.com>
 Dhruv Bhanushali <dhruv_b@live.com>
 Anjul Kumar Tyagi <anjul.ten@gmail.com>
 Barun Parruck <barun.parruck@gmail.com>
 Bao Chau <chauquocbao0907@gmail.com>
 Tanay Agrawal <tanay_agrawal@hotmail.com>
 Ranjith Kumar <ranjith.dakshana2015@gmail.com>
 Shikhar Makhija <shikharmakhija2@gmail.com>
 Yathartha Joshi <yathartha32@gmail.com>
 Valeriia Gladkova <valeriia.gladkova@gmail.com>
 Sagar Bharadwaj <sagarbharadwaj50@gmail.com>
 Daniel Mahler <dmahler@gmail.com>
 Ka Yi <chua.kayi@yahoo.com.sg>
 Rishat Iskhakov <iskhakov@frtk.ru>
 Szymon Mieszczyk <szymon.mieszczyk@gmail.com>
 Sachin Agarwal <sachin13agarwal@gmail.com>
 Priyank Patel <pspbot7@gmail.com>
 Satya Prakash Dwibedi <akash581050@gmail.com>
 tools4origins <tools4origins@gmail.com>
 Nico Schlömer <nico.schloemer@gmail.com>
 Fermi Paradox <FermiParadox@users.noreply.github.com>
 Ekansh Purohit <purohit.e15@gmail.com>
 Vedarth Sharma <vedarth.sharma@gmail.com>
 Peeyush Kushwaha <peeyush.p97@gmail.com>
 Jayjayyy <vfhsln8s3l4b87t4c3@byom.de>
 Christopher J. Wright <cjwright4242gh@gmail.com>
 Jakub Wilk <jwilk@jwilk.net>

(continues on next page)

(continued from previous page)

Mauro Garavello <mauro.garavello@unimib.it>
Chris Tefer <ctefer@gmail.com>
Shikhar Jaiswal <jaiswalshikhar87@gmail.com>
Chiu-Hsiang Hsu <wdv4758h@gmail.com>
Carlos Cordoba <ccordoba12@gmail.com>
Fabian Ball <fabian.ball@kit.edu>
Yerniyaz <yerniyaz.nurgabylov@nu.edu.kz>
Christiano Anderson <canderson@riseup.net>
Robin Neatherway <robin.neatherway@gmail.com>
Thomas Hunt <thomashunt13@gmail.com>
Theodore Han <theodorehan@hotmail.com>
Duc-Minh Phan <alephvn@gmail.com>
Lejla Metohajrova <l.metohajrova@gmail.com>
Samyak Jain <samyak.jain2016a@vitstudent.ac.in>
Aditya Rohan <riyuzakiiitk@gmail.com>
Vincent Delecroix <vincent.delecroix@labri.fr>
Michael Sparapany <msparapa@purdue.edu>
Harsh Jain <harshjniitr@gmail.com>
Nathan Goldbaum <ngoldebau@illinois.edu>
latot <felipematas@yahoo.com>
Kenneth Lyons <ixjlyons@gmail.com>
Stan Schymanski <stan.schymanski@env.ethz.ch>
David Daly <david.daly12@kzoo.edu>
Ayush Shridhar <ayush.shridhar1999@gmail.com>
Javed Nissar <javednissar@gmail.com>
Jiri Kuncar <jiri.kuncar@gmail.com>
vedantc98 <vedantc98@gmail.com>
Rupesh Harode <rupeshharode@gmail.com>
Rob Zinkov <rob@zinkov.com>
James Harrop <ebc121@gmail.com>
James Taylor <user234683@tutanota.com>
Ishan Joshi <ishanaj98@gmail.com>
Marco Mancini <marco.mancini@obspm.fr>
Boris Ettinger <ettinger.boris@gmail.com>
Micah Fitch <micahscopes@gmail.com>
Daniel Wennberg <daniel.wennberg@gmail.com>
ylemkimon <ylemkimon@naver.com>
Akash Vaish <akash.9712@gmail.com>
Peter Enenkel <peter.enenkel+git@gmail.com>
Waldir Pimenta <waldyrrious@gmail.com>
Jithin D. George <jithindgeorge93@gmail.com>
Lev Chelyadinov <leval81777@gmail.com>
Lucas Wiman <lucas.wiman@gmail.com>
Rhea Parekh <rheaparekh12@gmail.com>
James Cotton <peabodyl24@gmail.com>
Robert Pollak <robert.pollak@posteo.net>
anca-mc <anca-mc@users.noreply.github.com>
Sourav Ghosh <souravghosh2197@gmail.com>
Jonathan Allan <jjallan@users.noreply.github.com>
Nikhil Pappu <nkhlpappu@gmail.com>
Ethan Ward <etkewa@gmail.com>
Cezary Marczak <zeddql@gmail.com>
dps7ud <dps7ud@virginia.edu>
Nilabja Bhattacharya <nilabja10201992@gmail.com>
Itay4 <31018228+Itay4@users.noreply.github.com>
Poom Chiarawongse <eight1911@gmail.com>

(continues on next page)

(continued from previous page)

Yang Yang <wdscxsj@gmail.com>
 Cavendish McKay <cmckay@tachycline.com>
 Bradley Gannon <bradley.m.gannon@gmail.com>
 B McG <bmcg0890@gmail.com>
 Rob Drynkin <rob.drynkin@gmail.com>
 Seth Ebner <murgrehk@gmail.com>
 Akash Kundu <sk.sayakkundu1997@gmail.com>
 Mark Jeromin <mark.jeromin@sysfrog.net>
 Roberto Díaz Pérez <r.r.1994a@gmail.com>
 Gleb Siroki <g.shiroki@gmail.com>
 Segev Finer <segev208@gmail.com>
 Alex Lubbock <code@alexlubbock.com>
 Ayodeji Ige <ayodeji18@outlook.com>
 Matthew Wardrop <matthew.wardrop@airbnb.com>
 Hugo <hugovk@users.noreply.github.com>
 Austin Palmer <ap4000@nyu.edu>
 der-blaue-elefant <github@kklein.de>
 Filip Gokstorp <filip@gokstorp.se>
 Yuki Matsuda <yuki.matsuda.w@gmail.com>
 Aaron Miller <acmiller273@gmail.com>
 Salil Vishnu Kapur <salilvishnukapur@gmail.com>
 Atharva Khare <khareatharva@gmail.com>
 Shubham Maheshwari <rmaheshwari05@gmail.com>
 Pavel Tkachenko <paveltkachenko@email.com>
 Ashish Kumar Gaurav <ashishkg0022@gmail.com>
 Rajeev Singh <rajs2010@gmail.com>
 Keno Goertz <keno@goertz-berlin.com>
 Lucas Gallindo <lgallindo@gmail.com>
 Himanshu <hs80941@gmail.com>
 David Menéndez Hurtado <david.menendez.hurtado@scilifelab.se>
 Amit Manchanda <amitdelhi1995@gmail.com>
 Rohit Jain <rohitjain3241@gmail.com>
 Jonathan A. Gross <jarthurgross@gmail.com>
 Unknown <kunda@scribus.net>
 Sayan Goswami <Sayan98@users.noreply.github.com>
 Subhash Saurabh <subhashsaurabh419@gmail.com>
 Rastislav Rabatin <rastislav.rabatin@gmail.com>
 Vishal <vishalg2235@gmail.com>
 Jeremy Gluck <jeremygluck@yahoo.com>
 Akshat Maheshwari <akshat14714@gmail.com>
 symbolique <symbolique@users.noreply.github.com>
 Saloni Jain <tosalonijain@gmail.com>
 Arighna Chakrabarty <arighna.chakrabarty100@gmail.com>
 Abhigyan Khaund <mail@abhigyan.xyz>
 Jashanpreet Singh <jashansingh.4398@gmail.com>
 Saurabh Agarwal <shourabh.agarwal@gmail.com>
 luz.paz <luzpaz@users.noreply.github.com>
 P. Sai Prasanth <psai.prasanth.min16@itbhu.ac.in>
 Nirmal Sarswat <nirmalsarswat400@gmail.com>
 Cristian Di Pietrantonio <cristiandipietrantonio@gmail.com>
 Ravi charan <ravicharan.vsp@gmail.com>
 Nityananda Gohain <nityanandagohain@gmail.com>
 Cédric Travelletti <cedrictravelletti@gmail.com>
 Nicholas Bollweg <nick.bollweg@gmail.com>
 Himanshu Ladia <hladial99811@gmail.com>
 eward <eward@sunbelt-medical.com>

(continues on next page)

(continued from previous page)

```
Adwait Baokar <adwaitbaokar18@gmail.com>
Mihail Tarigradschi <m.tarigradschi@gmail.com>
Saketh <alurusaisaketh@gmail.com>
rushyam <rushyamsonu@gmail.com>
sfoo <sfoohei@gmail.com>
Rahil Hastu <rahilhastu@gmail.com>
Zach Raines <raineszm@gmail.com>
Sidhant Nagpal <sidhantnagpal97@gmail.com>
czgdp1807 <gdp.1807@gmail.com>
Rishav Chakraborty <anonymousxyz@outlook.com>
Malkhan Singh <malkhansinghrathaur@gmail.com>
Joaquim Monserrat <qmonserrat@mailoo.org>
Mayank Singh <mayank.singh081997@gmail.com>
Rémy Léone <rleone@online.net>
Maxence Mayrand <35958639+maxencemayrand@users.noreply.github.com>
Nikoleta Glynatsi <GlynatsiNE@cardiff.ac.uk>
helo9 <helo9@users.noreply.github.com>
Ken Wakita <wakita@is.titech.ac.jp>
Carl Sandrock <carl.sandrock@up.ac.za>
Fredrik Eriksson <freeriks@student.chalmers.se>
Ian Swire <oversizedpenguin@gmail.com>
Bulat <daianovich@mail.ru>
Ehren Metcalfe <ehren.m@gmail.com>
Dmitry Savransky <dsavransky@gmail.com>
Kiyohito Yamazaki <kyamaz@openql.org>
Caley Finn <caleyreuben@gmail.com>
zhouzq-thu <zzq_890709@hotmail.com>
Alexander Pozdneev <pozdneev@users.noreply.github.com>
Wes Turner <50891+westurner@users.noreply.github.com>
JMSS-Unknown <31131631+JMSS-Unknown@users.noreply.github.com>
Arshdeep Singh <singh.arshdeep1999@gmail.com>
Stewart Wadsworth <stewart.wadsworth@gmail.com>
Jared Lumpe <mjlumpe@gmail.com>
cym1 <16437732+cym1@users.noreply.github.com>
```

You can find a brief history of SymPy in the [README](#).

13.2 Financial and Infrastructure Support

- **Google:** SymPy has received generous financial support from Google in various years through the [Google Summer of Code](#) program by providing stipends:
 - in 2007 for 5 students ([GSoC 2007](#))
 - in 2008 for 1 student ([GSoC 2008](#))
 - in 2009 for 5 students ([GSoC 2009](#))
 - in 2010 for 5 students ([GSoC 2010](#))
 - in 2011 for 9 students ([GSoC 2011](#))
 - in 2012 for 6 students ([GSoC 2012](#))
 - in 2013 for 7 students ([GSoC 2013](#))
 - in 2014 for 10 students ([GSoC 2014](#))
 - in 2015 for 7 students ([GSoC 2015](#))

- in 2016 for 9 students ([GSoC 2016](#))

Of these, we would like to thank these organizations for hosting GSoC students under their umbrella organizations:

- [Python Software Foundation \(PSF\)](#) has hosted various GSoC students over the years:
 - * 3 GSoC 2007 students (Brian, Robert and Jason)
 - * 1 GSoC 2008 student (Fredrik)
 - * 2 GSoC 2009 students (Freddie and Priit)
 - * 4 GSoC 2010 students (Aaron, Christian, Matthew and Øyvind)
 - * 6 GSoC 2015 students
 - * 1 GSoC 2016 student (James)
- [Portland State University \(PSU\)](#) has hosted following GSoC students:
 - * 1 student (Chris) in 2007
 - * 3 students (Aaron, Dale and Fabian) in 2009
 - * 1 student (Addison) in 2010
- [The Space Telescope Science Institute](#): STScI hosted 1 GSoC 2007 student (Mateusz)
- [The Ruby Science Foundation](#) has hosted following GSoC students:
 - * 1 student (Abinash) in 2015
 - * 1 student (Rajith) in 2016
- Several 13-17 year old pre-university students contributed as part of Google's [Code-In 2011](#). ([GCI 2011](#))
- [Simula Research Laboratory](#): supports Pearu Peterson work in SymPy/SymPy Core projects
- [GitHub](#) is providing us with development and collaboration tools

13.3 License

Unless stated otherwise, all files in the SymPy project, SymPy's webpage (and wiki), all images and all documentation including this User's Guide are licensed using the new BSD license:

Copyright (c) 2006-2018 SymPy Development Team

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of SymPy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

(continues on next page)

(continued from previous page)

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Patches taken from the Diofant project (<https://github.com/diofant/diofant>) are licensed as:

Copyright (c) 2006-2017 SymPy Development Team,
2013-2017 Sergey B Kirpichev

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- c. Neither the name of Diofant, or SymPy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Submodule taken from the multipledispatch project (<https://github.com/mrocklin/multipledispatch>) are licensed as:

Copyright (c) 2014 Matthew Rocklin

(continues on next page)

(continued from previous page)

All rights reserved.

Redistribution **and** use **in** source **and** binary forms, **with or** without modification, are permitted provided that the following conditions are met:

- a. Redistributions of source code must retain the above copyright notice, this **list** of conditions **and** the following disclaimer.
- b. Redistributions **in** binary form must reproduce the above copyright notice, this **list** of conditions **and** the following disclaimer **in** the documentation **and/or** other materials provided **with** the distribution.
- c. Neither the name of multipledispatch nor the names of its contributors may be used to endorse **or** promote products derived **from this** software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS **"AS IS"** AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CITING SYMPY

To cite SymPy in publications use

Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 <https://doi.org/10.7717/peerj-cs.103>

A BibTeX entry for LaTeX users is

```
@article{10.7717/peerj-cs.103,
  title = {SymPy: symbolic computing in Python},
  author = {Meurer, Aaron and Smith, Christopher P. and Paprocki, Mateusz and \v{C}ertík, Ondřej and Kirpichev, Sergey B. and Rocklin, Matthew and Kumar, AMIT and Ivanov, Sergiu and Moore, Jason K. and Singh, Sartaj and Rathnayake, Thilina and Vig, Sean and Granger, Brian E. and Muller, Richard P. and Bonazzi, Francesco and Gupta, Harsh and Vats, Shivam and Johansson, Fredrik and Pedregosa, Fabian and Curry, Matthew J. and Terrel, Andy R. and Rou\v{c}ka, \v{S}těpán and Saboo, Ashutosh and Fernando, Isuru and Kulal, Sumith and Cimrman, Robert and Scopatz, Anthony},
  year = 2017,
  month = jan,
  keywords = {Python, Computer algebra system, Symbolics},
  abstract = {
    SymPy is an open source computer algebra system written in pure Python.
    It is built with a focus on extensibility and ease of use, through both interactive
    and programmatic applications. These characteristics have led SymPy to become a
    popular symbolic library for the scientific Python ecosystem. This paper presents
    the architecture of SymPy, a description of its features, and a discussion of
    select submodules. The supplementary material provide additional examples and
    further outline details of the architecture and features of SymPy.
  },
  volume = 3,
  pages = {e103},
  journal = {PeerJ Computer Science},
  issn = {2376-5992},
  url = {https://doi.org/10.7717/peerj-cs.103},
  doi = {10.7717/peerj-cs.103}
}
```

SymPy is BSD licensed, so you are free to use it whatever you like, be it academic, commercial, creating forks or derivatives, as long as you copy the BSD statement if you redistribute it (see the LICENSE file for details). That said, although not required by the SymPy license, if it is convenient for you, please cite SymPy when using it in your work and also consider contributing all your changes back, so that we can incorporate it and all of us will benefit in

the end.

BIBLIOGRAPHY

- [R49] http://en.wikipedia.org/wiki/Negative_number
- [R50] http://en.wikipedia.org/wiki/Parity_%28mathematics%29
- [R51] http://en.wikipedia.org/wiki/Imaginary_number
- [R52] http://en.wikipedia.org/wiki/Composite_number
- [R53] http://en.wikipedia.org/wiki/Irrational_number
- [R54] http://en.wikipedia.org/wiki/Prime_number
- [R55] <http://en.wikipedia.org/wiki/Finite>
- [R56] <https://docs.python.org/3/library/math.html#math.isfinite>
- [R57] <http://docs.scipy.org/doc/numpy/reference/generated/numpy.isfinite.html>
- [R58] <http://en.wikipedia.org/wiki/Zero>
- [R59] http://en.wikipedia.org/wiki/1_%28number%29
- [R60] http://en.wikipedia.org/wiki/%E2%88%921_%28number%29
- [R61] http://en.wikipedia.org/wiki/One_half
- [R62] <http://en.wikipedia.org/wiki/NaN>
- [R63] <http://en.wikipedia.org/wiki/Infinity>
- [R64] http://en.wikipedia.org/wiki/E_%28mathematical_constant%29
- [R65] http://en.wikipedia.org/wiki/Imaginary_unit
- [R66] <http://en.wikipedia.org/wiki/Pi>
- [R67] http://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant
- [R68] http://en.wikipedia.org/wiki/Catalan%27s_constant
- [R69] http://en.wikipedia.org/wiki/Golden_ratio
- [R70] <http://en.wikipedia.org/wiki/Exponentiation>
- [R71] http://en.wikipedia.org/wiki/Exponentiation#Zero_to_the_power_of_zero
- [R72] http://en.wikipedia.org/wiki/Indeterminate_forms
- [R73] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The and operator coerces each side into a bool, returning the object itself

when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute `x > y > z`, with `x`, `y`, and `z` being Symbols, Python converts the statement (roughly) into these steps:

1. `x > y > z`
2. `(x > y)` and `(y > z)`
3. `(GreaterThanObject)` and `(y > z)`
4. `(GreaterThanObject.__nonzero__())` and `(y > z)`
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like `x > y > z` work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R74] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, `1 > 2 > 3` is evaluated by Python as `(1 > 2)` and `(2 > 3)`. The `and` operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute `x > y > z`, with `x`, `y`, and `z` being Symbols, Python converts the statement (roughly) into these steps:

1. `x > y > z`
2. `(x > y)` and `(y > z)`
3. `(GreaterThanObject)` and `(y > z)`
4. `(GreaterThanObject.__nonzero__())` and `(y > z)`
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like `x > y > z` work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R75] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, `1 > 2 > 3` is evaluated by Python as `(1 > 2)` and `(2 > 3)`. The `and` operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute `x > y > z`, with `x`, `y`, and `z` being Symbols, Python converts the statement (roughly) into these steps:

1. `x > y > z`

2. $(x > y)$ and $(y > z)$
3. `(GreaterThanObject)` and $(y > z)$
4. `(GreaterThanObject.__nonzero__())` and $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R76] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The `and` operator coerces each side into a `bool`, returning the object itself when it short-circuits. The `bool` of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with x , y , and z being `Symbols`, Python converts the statement (roughly) into these steps:

1. $x > y > z$
2. $(x > y)$ and $(y > z)$
3. `(GreaterThanObject)` and $(y > z)$
4. `(GreaterThanObject.__nonzero__())` and $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R29] Skiena, S. ‘Permutations.’ 1.1 in *Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica*. Reading, MA: Addison-Wesley, pp. 3-16, 1990.

[R30] Knuth, D. E. *The Art of Computer Programming, Vol. 4: Combinatorial Algorithms*, 1st ed. Reading, MA: Addison-Wesley, 2011.

[R31] Wendy Myrvold and Frank Ruskey. 2001. Ranking and unranking permutations in linear time. *Inf. Process. Lett.* 79, 6 (September 2001), 281-284. DOI=10.1016/S0020-0190(01)00141-7

[R32] D. L. Kreher, D. R. Stinson ‘*Combinatorial Algorithms*’ CRC Press, 1999

[R33] Graham, R. L.; Knuth, D. E.; and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA: Addison-Wesley, 1994.

[R34] http://en.wikipedia.org/wiki/Permutation#Product_and_inverse

[R35] http://en.wikipedia.org/wiki/Lehmer_code

[R36] <http://mathworld.wolfram.com/LabeledTree.html>

- [CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463– 490, 1973.
- [Ho05] Derek F. Holt, *Handbook of Computational Group Theory*. In the series ‘Discrete Mathematics and its Applications’, *Chapman & Hall/CRC* 2005, xvi + 514 p.
- [Hav91] George Havas, Coset enumeration strategies. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC’91)*, Bonn 1991, pages 191–199. *ACM Press*, 1991.
- [R394] definition is described in <http://oeis.org/A066272/a066272a.html>
- [R395] formula from <https://oeis.org/A066272>
- [R396] https://en.wikipedia.org/wiki/Euler%27s_totient_function
- [R397] <http://mathworld.wolfram.com/TotientFunction.html>
- [R398] https://en.wikipedia.org/wiki/Carmichael_function
- [R399] <http://mathworld.wolfram.com/CarmichaelFunction.html>
- [R400] http://en.wikipedia.org/wiki/Divisor_function
- [R401] <http://mathworld.wolfram.com/UnitaryDivisorFunction.html>
- [R402] http://en.wikipedia.org/wiki/Square-free_integer#Squarefree_core
- [R403] <http://mathworld.wolfram.com/PrimeFactor.html>
- [R404] <http://mathworld.wolfram.com/PrimeFactor.html>
- [R405] <http://mathworld.wolfram.com/PartitionFunctionP.html>
- [R406] 23. Stein “Elementary Number Theory” (2011), page 44
- [R407] 16. Hackman “Elementary Number Theory” (2009), Chapter C
- [R408] 16. Hackman “Elementary Number Theory” (2009), page 76
- [R409] <http://mathworld.wolfram.com/DiscreteLogarithm.html>
- [R410] “Handbook of applied cryptography”, Menezes, A. J., Van, O. P. C., & Vanstone, S. A. (1997).
- [R411] http://en.wikipedia.org/wiki/Continued_fraction
- [R412] http://en.wikipedia.org/wiki/Periodic_continued_fraction
- [R413] K. Rosen. *Elementary Number theory and its applications*. Addison-Wesley, 3 Sub edition, pages 379-381, January 1992.
- [R414] http://en.wikipedia.org/wiki/M%C3%B6bius_function
- [R415] Thomas Koshy “Elementary Number Theory with Applications”
- [R416] http://en.wikipedia.org/wiki/Egyptian_fraction
- [R417] https://en.wikipedia.org/wiki/Greedy_algorithm_for_Egyptian_fractions
- [R418] <http://www.ics.uci.edu/~eppstein/numth/egypt/conflict.html>
- [R419] http://ami.ektf.hu/uploads/papers/finalpdf/AMI_42_from129to134.pdf
- [R77] http://en.wikipedia.org/wiki/Vigenere_cipher
- [R78] <http://web.archive.org/web/20071116100808/http://filebox.vt.edu/users/batman/kryptos.html> (short URL: <https://goo.gl/ijr22d>)
- [R79] en.wikipedia.org/wiki/Hill_cipher
- [R80] Lester S. Hill, *Cryptography in an Algebraic Alphabet*, *The American Mathematical Monthly* Vol.36, June-July 1929, pp.306-312.

- [R81] http://en.wikipedia.org/wiki/Morse_code
- [R82] http://en.wikipedia.org/wiki/Morse_code
- [G82] Solomon Golomb, Shift register sequences, Aegean Park Press, Laguna Hills, Ca, 1967
- [M83] James L. Massey, "Shift-Register Synthesis and BCH Decoding." IEEE Trans. on Information Theory, vol. 15(1), pp. 122-127, Jan 1969.
- [R37] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R38] http://en.wikipedia.org/wiki/Summation#Capital-sigma_notation
- [R39] http://en.wikipedia.org/wiki/Empty_sum
- [R40] https://en.wikipedia.org/wiki/Absolute_convergence
- [R41] https://en.wikipedia.org/wiki/Convergence_tests
- [R42] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R43] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R44] http://en.wikipedia.org/wiki/Multiplication#Capital_Pi_notation
- [R45] http://en.wikipedia.org/wiki/Empty_product
- [R46] https://en.wikipedia.org/wiki/Infinite_product
- [R47] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R48] Marko Petkovsek, Herbert S. Wilf, Doron Zeilberger, A = B, AK Peters, Ltd., Wellesley, MA, USA, 1997, pp. 73-100
- [R25] <https://docs.scipy.org/doc/numpy/user/basics.types.html>
- [R26] https://en.wikipedia.org/wiki/Newton%27s_method
- [R116] http://en.wikipedia.org/wiki/Complex_conjugation
- [R117] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R118] <http://dlmf.nist.gov/4.14>
- [R119] <http://functions.wolfram.com/ElementaryFunctions/Sin>
- [R120] <http://mathworld.wolfram.com/TrigonometryAngles.html>
- [R121] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R122] <http://dlmf.nist.gov/4.14>
- [R123] <http://functions.wolfram.com/ElementaryFunctions/Cos>
- [R124] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R125] <http://dlmf.nist.gov/4.14>
- [R126] <http://functions.wolfram.com/ElementaryFunctions/Tan>
- [R127] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R128] <http://dlmf.nist.gov/4.14>
- [R129] <http://functions.wolfram.com/ElementaryFunctions/Cot>
- [R130] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R131] <http://dlmf.nist.gov/4.14>

- [R132] <http://functions.wolfram.com/ElementaryFunctions/Sec>
- [R133] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R134] <http://dlmf.nist.gov/4.14>
- [R135] <http://functions.wolfram.com/ElementaryFunctions/Csc>
- [R136] http://en.wikipedia.org/wiki/Sinc_function
- [R137] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R138] <http://dlmf.nist.gov/4.23>
- [R139] <http://functions.wolfram.com/ElementaryFunctions/ArcSin>
- [R140] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R141] <http://dlmf.nist.gov/4.23>
- [R142] <http://functions.wolfram.com/ElementaryFunctions/ArcCos>
- [R143] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R144] <http://dlmf.nist.gov/4.23>
- [R145] <http://functions.wolfram.com/ElementaryFunctions/ArcTan>
- [R146] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R147] <http://dlmf.nist.gov/4.23>
- [R148] <http://functions.wolfram.com/ElementaryFunctions/ArcCot>
- [R149] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R150] <http://dlmf.nist.gov/4.23>
- [R151] <http://functions.wolfram.com/ElementaryFunctions/ArcSec>
- [R152] <http://reference.wolfram.com/language/ref/ArcSec.html>
- [R153] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R154] <http://dlmf.nist.gov/4.23>
- [R155] <http://functions.wolfram.com/ElementaryFunctions/ArcCsc>
- [R156] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R157] <http://en.wikipedia.org/wiki/Atan2>
- [R158] <http://functions.wolfram.com/ElementaryFunctions/ArcTan2>
- [R159] http://en.wikipedia.org/wiki/Hyperbolic_function
- [R160] <http://dlmf.nist.gov/4.37>
- [R161] <http://functions.wolfram.com/ElementaryFunctions/ArcSech/>
- [R162] http://en.wikipedia.org/wiki/Hyperbolic_function
- [R163] <http://dlmf.nist.gov/4.37>
- [R164] <http://functions.wolfram.com/ElementaryFunctions/ArcCsch/>
- [R165] “Concrete mathematics” by Graham, pp. 87
- [R166] <http://mathworld.wolfram.com/CeilingFunction.html>
- [R167] “Concrete mathematics” by Graham, pp. 87
- [R168] <http://mathworld.wolfram.com/FloorFunction.html>
- [R169] http://en.wikipedia.org/wiki/Fractional_part

- [R170] <http://mathworld.wolfram.com/FractionalPart.html>
- [R171] http://en.wikipedia.org/wiki/Lambert_W_function
- [R172] http://en.wikipedia.org/wiki/Directed_complete_partial_order
- [R173] http://en.wikipedia.org/wiki/Lattice_%28order%29
- [R174] http://en.wikipedia.org/wiki/Square_root
- [R175] http://en.wikipedia.org/wiki/Principal_value
- [R85] http://en.wikipedia.org/wiki/Bell_number
- [R86] <http://mathworld.wolfram.com/BellNumber.html>
- [R87] <http://mathworld.wolfram.com/BellPolynomial.html>
- [R88] http://en.wikipedia.org/wiki/Bernoulli_number
- [R89] http://en.wikipedia.org/wiki/Bernoulli_polynomial
- [R90] <http://mathworld.wolfram.com/BernoulliNumber.html>
- [R91] <http://mathworld.wolfram.com/BernoulliPolynomial.html>
- [R92] https://www.johndcook.com/blog/binomial_coefficients/
- [R93] http://en.wikipedia.org/wiki/Catalan_number
- [R94] <http://mathworld.wolfram.com/CatalanNumber.html>
- [R95] <http://functions.wolfram.com/GammaBetaErf/CatalanNumber/>
- [R96] <http://geometer.org/mathcircles/catalan.pdf>
- [R97] http://en.wikipedia.org/wiki/Euler_numbers
- [R98] <http://mathworld.wolfram.com/EulerNumber.html>
- [R99] http://en.wikipedia.org/wiki/Alternating_permutation
- [R100] <http://mathworld.wolfram.com/AlternatingPermutation.html>
- [R101] <http://dlmf.nist.gov/24.2#ii>
- [R102] <http://en.wikipedia.org/wiki/Subfactorial>
- [R103] <http://mathworld.wolfram.com/Subfactorial.html>
- [R104] https://en.wikipedia.org/wiki/Double_factorial
- [R105] <http://mathworld.wolfram.com/FallingFactorial.html>
- [R106] http://en.wikipedia.org/wiki/Fibonacci_number
- [R107] <http://mathworld.wolfram.com/FibonacciNumber.html>
- [R108] http://en.wikipedia.org/wiki/Harmonic_number
- [R109] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/>
- [R110] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/>
- [R111] http://en.wikipedia.org/wiki/Lucas_number
- [R112] <http://mathworld.wolfram.com/LucasNumber.html>
- [R113] https://en.wikipedia.org/wiki/Pochhammer_symbol
- [R114] http://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind
- [R115] http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind
- [R176] <http://mathworld.wolfram.com/DeltaFunction.html>

- [R177] Regarding to the value at 0, Mathematica defines $H(0) = 1$, but Maple uses $H(0) = \text{undefined}$. Different application areas may have specific conventions. For example, in control theory, it is common practice to assume $H(0) == 0$ to match the Laplace transform of a DiracDelta distribution.
- [R178] <http://mathworld.wolfram.com/HeavisideStepFunction.html>
- [R179] <http://dlmf.nist.gov/1.16#iv>
- [R180] https://en.wikipedia.org/wiki/Singularity_function
- [R181] http://en.wikipedia.org/wiki/Gamma_function
- [R182] <http://dlmf.nist.gov/5>
- [R183] <http://mathworld.wolfram.com/GammaFunction.html>
- [R184] <http://functions.wolfram.com/GammaBetaErf/Gamma/>
- [R185] http://en.wikipedia.org/wiki/Gamma_function
- [R186] <http://dlmf.nist.gov/5>
- [R187] <http://mathworld.wolfram.com/LogGammaFunction.html>
- [R188] <http://functions.wolfram.com/GammaBetaErf/LogGamma/>
- [R189] http://en.wikipedia.org/wiki/Polygamma_function
- [R190] <http://mathworld.wolfram.com/PolygammaFunction.html>
- [R191] <http://functions.wolfram.com/GammaBetaErf/PolyGamma/>
- [R192] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R193] http://en.wikipedia.org/wiki/Digamma_function
- [R194] <http://mathworld.wolfram.com/DigammaFunction.html>
- [R195] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R196] http://en.wikipedia.org/wiki/Trigamma_function
- [R197] <http://mathworld.wolfram.com/TrigammaFunction.html>
- [R198] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R199] http://en.wikipedia.org/wiki/Incomplete_gamma_function#Upper_incomplete_Gamma_function
- [R200] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R201] <http://dlmf.nist.gov/8>
- [R202] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R203] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R204] http://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions
- [R205] http://en.wikipedia.org/wiki/Incomplete_gamma_function#Lower_incomplete_Gamma_function
- [R206] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R207] <http://dlmf.nist.gov/8>
- [R208] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R209] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>

- [R210] http://en.wikipedia.org/wiki/Beta_function
- [R211] <http://mathworld.wolfram.com/BetaFunction.html>
- [R212] <http://dlmf.nist.gov/5.12>
- [R213] http://en.wikipedia.org/wiki/Error_function
- [R214] <http://dlmf.nist.gov/7>
- [R215] <http://mathworld.wolfram.com/Erf.html>
- [R216] <http://functions.wolfram.com/GammaBetaErf/Erf>
- [R217] http://en.wikipedia.org/wiki/Error_function
- [R218] <http://dlmf.nist.gov/7>
- [R219] <http://mathworld.wolfram.com/Erfc.html>
- [R220] <http://functions.wolfram.com/GammaBetaErf/Erfc>
- [R221] http://en.wikipedia.org/wiki/Error_function
- [R222] <http://mathworld.wolfram.com/Erfi.html>
- [R223] <http://functions.wolfram.com/GammaBetaErf/Erfi>
- [R224] <http://functions.wolfram.com/GammaBetaErf/Erf2/>
- [R225] http://en.wikipedia.org/wiki/Error_function#Inverse_functions
- [R226] <http://functions.wolfram.com/GammaBetaErf/InverseErf/>
- [R227] http://en.wikipedia.org/wiki/Error_function#Inverse_functions
- [R228] <http://functions.wolfram.com/GammaBetaErf/InverseErfc/>
- [R229] <http://functions.wolfram.com/GammaBetaErf/InverseErf2/>
- [R230] http://en.wikipedia.org/wiki/Fresnel_integral
- [R231] <http://dlmf.nist.gov/7>
- [R232] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R233] <http://functions.wolfram.com/GammaBetaErf/FresnelS>
- [R234] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R235] http://en.wikipedia.org/wiki/Fresnel_integral
- [R236] <http://dlmf.nist.gov/7>
- [R237] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R238] <http://functions.wolfram.com/GammaBetaErf/FresnelC>
- [R239] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R240] <http://dlmf.nist.gov/6.6>
- [R241] http://en.wikipedia.org/wiki/Exponential_integral
- [R242] Abramowitz & Stegun, section 5: http://people.math.sfu.ca/~cbm/aands/page_228.htm
- [R243] <http://dlmf.nist.gov/8.19>
- [R244] <http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/>
- [R245] http://en.wikipedia.org/wiki/Exponential_integral
- [R246] http://en.wikipedia.org/wiki/Logarithmic_integral

- [R247] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R248] <http://dlmf.nist.gov/6>
- [R249] <http://mathworld.wolfram.com/SoldnersConstant.html>
- [R250] http://en.wikipedia.org/wiki/Logarithmic_integral
- [R251] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R252] <http://dlmf.nist.gov/6>
- [R253] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R254] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R255] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R256] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R257] Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 9", Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R258] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R259] http://en.wikipedia.org/wiki/Bessel_function
- [R260] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/>
- [R261] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/>
- [R262] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/>
- [R263] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/>
- [R264] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/>
- [R265] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/>
- [R266] <http://dlmf.nist.gov/10.47>
- [R267] <http://dlmf.nist.gov/10.47>
- [R268] http://en.wikipedia.org/wiki/Airy_function
- [R269] <http://dlmf.nist.gov/9>
- [R270] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R271] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R272] http://en.wikipedia.org/wiki/Airy_function
- [R273] <http://dlmf.nist.gov/9>
- [R274] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R275] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R276] http://en.wikipedia.org/wiki/Airy_function
- [R277] <http://dlmf.nist.gov/9>
- [R278] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R279] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R280] http://en.wikipedia.org/wiki/Airy_function
- [R281] <http://dlmf.nist.gov/9>
- [R282] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R283] <http://mathworld.wolfram.com/AiryFunctions.html>

- [R284] <http://en.wikipedia.org/wiki/B-spline>
- [R285] <http://dlmf.nist.gov/25.11>
- [R286] http://en.wikipedia.org/wiki/Hurwitz_zeta_function
- [R287] http://en.wikipedia.org/wiki/Dirichlet_eta_function
- [R288] Bateman, H.; Erdelyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill. Section 1.11.
- [R289] <http://dlmf.nist.gov/25.14>
- [R290] http://en.wikipedia.org/wiki/Lerch_transcendent
- [R291] http://en.wikipedia.org/wiki/Stieltjes_constants
- [R292] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R293] http://en.wikipedia.org/wiki/Generalized_hypergeometric_function
- [R294] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R295] http://en.wikipedia.org/wiki/Meijer_G-function
- [R296] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R297] <http://functions.wolfram.com/EllipticIntegrals/EllipticK>
- [R298] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R299] <http://functions.wolfram.com/EllipticIntegrals/EllipticF>
- [R300] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R301] <http://functions.wolfram.com/EllipticIntegrals/EllipticE2>
- [R302] <http://functions.wolfram.com/EllipticIntegrals/EllipticE>
- [R303] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R304] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi3>
- [R305] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi>
- [R306] http://en.wikipedia.org/wiki/Mathieu_function
- [R307] <http://dlmf.nist.gov/28>
- [R308] <http://mathworld.wolfram.com/MathieuBase.html>
- [R309] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuS/>
- [R310] http://en.wikipedia.org/wiki/Mathieu_function
- [R311] <http://dlmf.nist.gov/28>
- [R312] <http://mathworld.wolfram.com/MathieuBase.html>
- [R313] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuC/>
- [R314] http://en.wikipedia.org/wiki/Mathieu_function
- [R315] <http://dlmf.nist.gov/28>
- [R316] <http://mathworld.wolfram.com/MathieuBase.html>
- [R317] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuSPRime/>
- [R318] http://en.wikipedia.org/wiki/Mathieu_function
- [R319] <http://dlmf.nist.gov/28>
- [R320] <http://mathworld.wolfram.com/MathieuBase.html>

- [R321] <http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuCPrime/>
- [R322] http://en.wikipedia.org/wiki/Jacobi_polynomials
- [R323] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R324] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R325] http://en.wikipedia.org/wiki/Jacobi_polynomials
- [R326] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R327] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R328] http://en.wikipedia.org/wiki/Gegenbauer_polynomials
- [R329] <http://mathworld.wolfram.com/GegenbauerPolynomial.html>
- [R330] <http://functions.wolfram.com/Polynomials/GegenbauerC3/>
- [R331] http://en.wikipedia.org/wiki/Chebyshev_polynomial
- [R332] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R333] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R334] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R335] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R336] http://en.wikipedia.org/wiki/Chebyshev_polynomial
- [R337] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R338] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R339] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R340] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R341] http://en.wikipedia.org/wiki/Legendre_polynomial
- [R342] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R343] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R344] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R345] http://en.wikipedia.org/wiki/Associated_Legendre_polynomials
- [R346] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R347] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R348] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R349] http://en.wikipedia.org/wiki/Hermite_polynomial
- [R350] <http://mathworld.wolfram.com/HermitePolynomial.html>
- [R351] <http://functions.wolfram.com/Polynomials/HermiteH/>
- [R352] http://en.wikipedia.org/wiki/Laguerre_polynomial
- [R353] <http://mathworld.wolfram.com/LaguerrePolynomial.html>
- [R354] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R355] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R356] http://en.wikipedia.org/wiki/Laguerre_polynomial#Assoc_laguerre_polynomials
- [R357] <http://mathworld.wolfram.com/AssociatedLaguerrePolynomial.html>
- [R358] <http://functions.wolfram.com/Polynomials/LaguerreL/>

- [R359] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R360] http://en.wikipedia.org/wiki/Spherical_harmonics
- [R361] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R362] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R363] <http://dlmf.nist.gov/14.30>
- [R364] http://en.wikipedia.org/wiki/Spherical_harmonics
- [R365] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R366] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R367] http://en.wikipedia.org/wiki/Spherical_harmonics
- [R368] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R369] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R370] http://en.wikipedia.org/wiki/Kronecker_delta
- [WikiPappus] “Pappus’s Hexagon Theorem” Wikipedia, the Free Encyclopedia. Web. 26 Apr. 2013. <http://en.wikipedia.org/wiki/Pappus's_hexagon_theorem>
- [R371] <https://hal.inria.fr/inria-00070025/document>
- [R372] http://www.risc.jku.at/publications/download/risc_2244/DIPLFORM.pdf
- [BlogPost] <http://nessgrh.wordpress.com/2011/07/07/tricky-branch-cuts/>
- [R373] http://en.wikipedia.org/wiki/Gaussian_quadrature
- [R374] http://people.sc.fsu.edu/~jburkardt/cpp_src/legendre_rule/legendre_rule.html
- [R375] http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature
- [R376] http://people.sc.fsu.edu/~jburkardt/cpp_src/laguerre_rule/laguerre_rule.html
- [R377] http://en.wikipedia.org/wiki/Gauss-Hermite_Quadrature
- [R378] http://people.sc.fsu.edu/~jburkardt/cpp_src/hermite_rule/hermite_rule.html
- [R379] http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_hermite_rule/gen_hermite_rule.html
- [R380] http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature
- [R381] http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_laguerre_rule/gen_laguerre_rule.html
- [R382] http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature
- [R383] http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev1_rule/chebyshev1_rule.html
- [R384] http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature
- [R385] http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev2_rule/chebyshev2_rule.html
- [R386] http://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature
- [R387] http://people.sc.fsu.edu/~jburkardt/cpp_src/jacobi_rule/jacobi_rule.html
- [R388] http://people.sc.fsu.edu/~jburkardt/cpp_src/gegenbauer_rule/gegenbauer_rule.html
- [R389] en.wikipedia.org/wiki/Quine-McCluskey_algorithm
- [R390] en.wikipedia.org/wiki/Quine-McCluskey_algorithm
- [R391] http://en.wikipedia.org/wiki/Gaussian_elimination
- [R392] https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse

- [R393] https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse#Obtaining_all_solutions_of_a_linear_system
- [Wester1999] Michael J. Wester, A Critique of the Mathematical Abilities of CA Systems, 1999, <http://www.math.unm.edu/~wester/cas/book/Wester.pdf>
- [Kozen89] D. Kozen, S. Landau, Polynomial decomposition algorithms, Journal of Symbolic Computation 7 (1989), pp. 445-456
- [Liao95] Hsin-Chao Liao, R. Fateman, Evaluation of the heuristic polynomial GCD, International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, Montreal, Quebec, Canada, 1995, pp. 240-247
- [Gathen99] J. von zur Gathen, J. Gerhard, Modern Computer Algebra, First Edition, Cambridge University Press, 1999
- [Weisstein09] Eric W. Weisstein, Cyclotomic Polynomial, From MathWorld - A Wolfram Web Resource, <http://mathworld.wolfram.com/CyclotomicPolynomial.html>
- [Wang78] P. S. Wang, An Improved Multivariate Polynomial Factoring Algorithm, Math. of Computation 32, 1978, pp. 1215-1231
- [Geddes92] K. Geddes, S. R. Czapor, G. Labahn, Algorithms for Computer Algebra, Springer, 1992
- [Monagan93] Michael Monagan, In-place Arithmetic for Polynomials over $\mathbb{Z}_n$, Proceedings of DISCO '92, Springer-Verlag LNCS, 721, 1993, pp. 22-34
- [Kaltofen98] E. Kaltofen, V. Shoup, Subquadratic-time Factoring of Polynomials over Finite Fields, Mathematics of Computation, Volume 67, Issue 223, 1998, pp. 1179-1197
- [Shoup95] V. Shoup, A New Polynomial Factorization Algorithm and its Implementation, Journal of Symbolic Computation, Volume 20, Issue 4, 1995, pp. 363-397
- [Gathen92] J. von zur Gathen, V. Shoup, Computing Frobenius Maps and Factoring Polynomials, ACM Symposium on Theory of Computing, 1992, pp. 187-224
- [Shoup91] V. Shoup, A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic, In Proceedings of International Symposium on Symbolic and Algebraic Computation, 1991, pp. 14-21
- [Cox97] D. Cox, J. Little, D. O'Shea, Ideals, Varieties and Algorithms, Springer, Second Edition, 1997
- [Ajwa95] I.A. Ajwa, Z. Liu, P.S. Wang, Groebner Bases Algorithm, <https://citeseer.ist.psu.edu/myciteseer/login>, 1995
- [Bose03] N.K. Bose, B. Buchberger, J.P. Guiver, Multidimensional Systems Theory and Applications, Springer, 2003
- [Giovini91] A. Giovini, T. Mora, "One sugar cube, please" or Selection strategies in Buchberger algorithm, ISSAC '91, ACM
- [Bronstein93] M. Bronstein, B. Salvy, Full partial fraction decomposition of rational functions, Proceedings ISSAC '93, ACM Press, Kiev, Ukraine, 1993, pp. 157-160
- [Buchberger01] B. Buchberger, Groebner Bases: A Short Introduction for Systems Theorists, In: R. Moreno-Diaz, B. Buchberger, J. L. Freire, Proceedings of EUROCAST'01, February, 2001
- [Davenport88] J.H. Davenport, Y. Siret, E. Tournier, Computer Algebra Systems and Algorithms for Algebraic Computation, Academic Press, London, 1988, pp. 124-128
- [Greuel2008] G.-M. Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer, 2008

- [Atiyah69] M.F. Atiyah, I.G. MacDonald, Introduction to Commutative Algebra, Addison-Wesley, 1969
- [Collins67] G.E. Collins, Subresultants and Reduced Polynomial Remainder Sequences. J. ACM 14 (1967) 128-142
- [BrownTraub71] W.S. Brown, J.F. Traub, On Euclid's Algorithm and the Theory of Subresultants. J. ACM 18 (1971) 505-514
- [Brown78] W.S. Brown, The Subresultant PRS Algorithm. ACM Transaction of Mathematical Software 4 (1978) 237-249
- [Monagan00] M. Monagan and A. Wittkopf, On the Design and Implementation of Brown's Algorithm over the Integers and Number Fields, Proceedings of ISSAC 2000, pp. 225-233, ACM, 2000.
- [Brown71] W.S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, J. ACM 18, 4, pp. 478-504, 1971.
- [Hoeij04] M. van Hoeij and M. Monagan, Algorithms for polynomial GCD computation over algebraic function fields, Proceedings of ISSAC 2004, pp. 297-304, ACM, 2004.
- [Wang81] P.S. Wang, A p-adic algorithm for univariate partial fractions, Proceedings of SYMSAC 1981, pp. 212-217, ACM, 1981.
- [Hoeij02] M. van Hoeij and M. Monagan, A modular GCD algorithm over number fields presented with multiple extensions, Proceedings of ISSAC 2002, pp. 109-116, ACM, 2002
- [ManWright94] Yiu-Kwong Man and Francis J. Wright, "Fast Polynomial Dispersion Computation and its Application to Indefinite Summation", Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1994, Pages 175-180 <http://dl.acm.org/citation.cfm?doid=190347.190413>
- [Koepf98] Wolfram Koepf, "Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities", Advanced lectures in mathematics, Vieweg, 1998
- [Abramov71] S. A. Abramov, "On the Summation of Rational Functions", USSR Computational Mathematics and Mathematical Physics, Volume 11, Issue 4, 1971, Pages 324-330
- [Man93] Yiu-Kwong Man, "On Computing Closed Forms for Indefinite Summations", Journal of Symbolic Computation, Volume 16, Issue 4, 1993, Pages 355-376 <http://www.sciencedirect.com/science/article/pii/S0747717183710539>
- [Kapur1994] Deepak Kapur, Tushar Saxena, and Lu Yang. "Algebraic and geometric reasoning using Dixon resultants", In Proceedings of the international symposium on Symbolic and algebraic computation (ISSAC '94), 1994, pages 99-107. https://www.researchgate.net/publication/2514261_Algebraic_and_Geometric_Reasoning_using_Dixon_Resultants
- [Palancz08] B Paláncz, P Zaletnyik, JL Awange, EW Grafarend. "Dixon resultant's solution of systems of geodetic polynomial equations", Journal of Geodesy, 2008, Springer, https://www.researchgate.net/publication/225607735_Dixon_resultant's_solution_of_systems_of_geodetic_polynomial_equations
- [Bruce97] Bruce Randall Donald, Deepak Kapur, and Joseph L. Mundy (Eds.). "Symbolic and Numerical Computation for Artificial Intelligence", Chapter 2, Academic Press, Inc., Orlando, FL, USA, 1997, <https://www2.cs.duke.edu/donaldlab/Books/SymbolicNumericalComputation/045-087.pdf>.
- [Stiller96] P Stiller. "An introduction to the theory of resultants", Mathematics and Computer Science, T&M University, 1996, Citeseer, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.590.2021&rep=rep1&type=pdf>.
- [R1] http://en.wikipedia.org/wiki/Algebraic_number
- [R2] <http://mathworld.wolfram.com/HermitianOperator.html>

- [R3] https://en.wikipedia.org/wiki/Complex_number
- [R4] https://en.wikipedia.org/wiki/Diagonal_matrix
- [R5] <https://en.wikipedia.org/wiki/Finite>
- [R6] <http://mathworld.wolfram.com/HermitianOperator.html>
- [R7] https://en.wikipedia.org/wiki/Imaginary_number
- [R8] <https://en.wikipedia.org/wiki/Integer>
- [R9] https://en.wikipedia.org/wiki/Invertible_matrix
- [R10] https://en.wikipedia.org/wiki/Irrational_number
- [R11] <http://mathworld.wolfram.com/LowerTriangularMatrix.html>
- [R12] https://en.wikipedia.org/wiki/Normal_matrix
- [R13] https://en.wikipedia.org/wiki/Orthogonal_matrix
- [R14] https://en.wikipedia.org/wiki/Positive-definite_matrix
- [R15] https://en.wikipedia.org/wiki/Real_number
- [R16] <http://mathworld.wolfram.com/SingularMatrix.html>
- [R17] https://en.wikipedia.org/wiki/Square_matrix
- [R18] https://en.wikipedia.org/wiki/Symmetric_matrix
- [R19] https://en.wikipedia.org/wiki/Triangular_matrix
- [R20] https://en.wikipedia.org/wiki/Unitary_matrix
- [R21] <http://mathworld.wolfram.com/UpperTriangularMatrix.html>
- [R457] Big O notation
- [R452] https://en.wikipedia.org/wiki/Gibbs_phenomenon
- [R453] https://en.wikipedia.org/wiki/Sigma_approximation
- [R454] mathworld.wolfram.com/FourierSeries.html
- [R448] Formal Power Series - Dominik Gruntz, Wolfram Koepf
- [R449] Power Series in Computer Algebra - Wolfram Koepf
- [R450] Formal Power Series - Dominik Gruntz, Wolfram Koepf
- [R451] Power Series in Computer Algebra - Wolfram Koepf
- [R455] <https://reference.wolfram.com/language/ref/DifferenceDelta.html>
- [R456] Computing Limits of Sequences - Manuel Kauers
- [R458] http://en.wikipedia.org/wiki/Disjoint_sets
- [R459] http://en.wikipedia.org/wiki/Power_set
- [R460] https://en.wikipedia.org/wiki/Symmetric_difference
- [R461] http://en.wikipedia.org/wiki/Interval_%28mathematics%29
- [R462] http://en.wikipedia.org/wiki/Finite_set
- [R463] http://en.wikipedia.org/wiki/Union_%28set_theory%29
- [R464] http://en.wikipedia.org/wiki/Intersection_%28set_theory%29
- [R465] http://en.wikipedia.org/wiki/Cartesian_product
- [R466] <http://mathworld.wolfram.com/ComplementSet.html>

- [R467] http://en.wikipedia.org/wiki/Empty_set
- [R468] http://en.wikipedia.org/wiki/Universal_set
- [Roach1996] Kelly B. Roach. Hypergeometric Function Representations. In: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, pages 301-308, New York, 1996. ACM.
- [Roach1997] Kelly B. Roach. Meijer G Function Representations. In: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, pages 205-211, New York, 1997. ACM.
- [Luke1969] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1.
- [Prudnikov1990] A. P. Prudnikov, Yu. A. Brychkov and O. I. Marichev (1990). Integrals and Series: More Special Functions, Vol. 3, Gordon and Breach Science Publisher.
- [R498] http://en.wikipedia.org/wiki/Arcsine_distribution
- [R499] http://en.wikipedia.org/wiki/Benini_distribution
- [R500] <http://reference.wolfram.com/legacy/v8/ref/BeniniDistribution.html>
- [R501] http://en.wikipedia.org/wiki/Beta_distribution
- [R502] <http://mathworld.wolfram.com/BetaDistribution.html>
- [R503] http://en.wikipedia.org/wiki/Beta_prime_distribution
- [R504] <http://mathworld.wolfram.com/BetaPrimeDistribution.html>
- [R505] http://en.wikipedia.org/wiki/Cauchy_distribution
- [R506] <http://mathworld.wolfram.com/CauchyDistribution.html>
- [R507] http://en.wikipedia.org/wiki/Chi_distribution
- [R508] <http://mathworld.wolfram.com/ChiDistribution.html>
- [R509] http://en.wikipedia.org/wiki/Noncentral_chi_distribution
- [R510] http://en.wikipedia.org/wiki/Chi_squared_distribution
- [R511] <http://mathworld.wolfram.com/Chi-SquaredDistribution.html>
- [R512] http://en.wikipedia.org/wiki/Dagum_distribution
- [R513] http://en.wikipedia.org/wiki/Erlang_distribution
- [R514] <http://mathworld.wolfram.com/ErlangDistribution.html>
- [R515] http://en.wikipedia.org/wiki/Exponential_distribution
- [R516] <http://mathworld.wolfram.com/ExponentialDistribution.html>
- [R517] <http://en.wikipedia.org/wiki/F-distribution>
- [R518] <http://mathworld.wolfram.com/F-Distribution.html>
- [R519] http://en.wikipedia.org/wiki/Fisher%27s_z-distribution
- [R520] <http://mathworld.wolfram.com/Fishersz-Distribution.html>
- [R521] http://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution
- [R522] http://en.wikipedia.org/wiki/Gamma_distribution
- [R523] <http://mathworld.wolfram.com/GammaDistribution.html>
- [R524] http://en.wikipedia.org/wiki/Inverse-gamma_distribution
- [R525] http://en.wikipedia.org/wiki/Kumaraswamy_distribution
- [R526] http://en.wikipedia.org/wiki/Laplace_distribution

- [R527] <http://mathworld.wolfram.com/LaplaceDistribution.html>
- [R528] http://en.wikipedia.org/wiki/Logistic_distribution
- [R529] <http://mathworld.wolfram.com/LogisticDistribution.html>
- [R530] <http://en.wikipedia.org/wiki/Lognormal>
- [R531] <http://mathworld.wolfram.com/LogNormalDistribution.html>
- [R532] http://en.wikipedia.org/wiki/Maxwell_distribution
- [R533] <http://mathworld.wolfram.com/MaxwellDistribution.html>
- [R534] http://en.wikipedia.org/wiki/Nakagami_distribution
- [R535] http://en.wikipedia.org/wiki/Normal_distribution
- [R536] <http://mathworld.wolfram.com/NormalDistributionFunction.html>
- [R537] http://en.wikipedia.org/wiki/Pareto_distribution
- [R538] <http://mathworld.wolfram.com/ParetoDistribution.html>
- [R539] http://en.wikipedia.org/wiki/U-quadratic_distribution
- [R540] http://en.wikipedia.org/wiki/Raised_cosine_distribution
- [R541] http://en.wikipedia.org/wiki/Rayleigh_distribution
- [R542] <http://mathworld.wolfram.com/RayleighDistribution.html>
- [R543] http://en.wikipedia.org/wiki/Student_t-distribution
- [R544] <http://mathworld.wolfram.com/Studentst-Distribution.html>
- [R545] http://en.wikipedia.org/wiki/Triangular_distribution
- [R546] <http://mathworld.wolfram.com/TriangularDistribution.html>
- [R547] http://en.wikipedia.org/wiki/Uniform_distribution_%28continuous%29
- [R548] <http://mathworld.wolfram.com/UniformDistribution.html>
- [R549] http://en.wikipedia.org/wiki/Uniform_sum_distribution
- [R550] <http://mathworld.wolfram.com/UniformSumDistribution.html>
- [R551] http://en.wikipedia.org/wiki/Von_Mises_distribution
- [R552] <http://mathworld.wolfram.com/vonMisesDistribution.html>
- [R553] http://en.wikipedia.org/wiki/Weibull_distribution
- [R554] <http://mathworld.wolfram.com/WeibullDistribution.html>
- [R555] http://en.wikipedia.org/wiki/Wigner_semicircle_distribution
- [R556] <http://mathworld.wolfram.com/WignersSemicircleLaw.html>
- [R492] S. A. Abramov, M. Bronstein and M. Petkovsek, On polynomial solutions of linear operator equations, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 290-296.
- [R493] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.
- [R494] 13. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996.
- [R495] S. A. Abramov, Rational solutions of linear difference and q-difference equations with polynomial coefficients, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 285-289
- [R496] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.

- [R497] 13. Petkovsek, H. S. Wilf, D. Zeilberger, $A = B$, 1996.
- [R469] Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>
- [R470] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <http://www.jpr2718.org/ax2p.pdf>
- [R471] Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Pages 16 - 17. [online], Available: <http://www.jpr2718.org/pell.pdf>
- [R472] 1. Nitaj, "L'algorithme de Cornacchia"
- [R473] Solving the diophantine equation $ax^2 + by^2 = m$ by Cornacchia's method, [online], Available: <http://www.numbertheory.org/php/cornacchia.html>
- [R474] Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Page 15. <http://www.jpr2718.org/pell.pdf>
- [R475] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, John P. Robertson, May 8, 2003, Page 7 - 11. <http://www.jpr2718.org/ax2p.pdf>
- [R476] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, John P. Robertson, May 8, 2003, Page 7 - 11. <http://www.jpr2718.org/ax2p.pdf>
- [R477] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R478] Representing an integer as a sum of three squares, [online], Available: http://www.proofwiki.org/wiki/Integer_as_Sum_of_Three_Squares
- [R479] Representing a number as a sum of three squares, [online], Available: <http://schorn.ch/lagrange.html>
- [R480] Representing a number as a sum of four squares, [online], Available: <http://schorn.ch/lagrange.html>
- [R481] Solving the generalized Pell equation $x^2 - Dy^2 = N$, John P. Robertson, July 31, 2004, Pages 4 - 8. <http://www.jpr2718.org/pell.pdf>
- [R482] Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Page 12. <http://www.jpr2718.org/pell.pdf>
- [R483] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R484] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R485] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, [online], Available: <http://eprints.nottingham.ac.uk/60/1/kvxefz87.pdf>
- [R486] Gaussian lattice Reduction [online]. Available: <http://home.ie.cuhk.edu.hk/~wkshum/wordpress/?p=404>
- [R487] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R488] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R489] Diophantine Equations, L. J. Mordell, page 48.
- [R490] Representing a number as a sum of four squares, [online], Available: <http://schorn.ch/lagrange.html>

- [R491] Legendre's Theorem, Legrange's Descent, http://public.csusm.edu/aitken_html/notes/legendre.pdf
- [AOCP] Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.
- [Factorisatio] On a Problem of Oppenheim concerning "Factorisatio Numerorum" E. R. Canfield, Paul Erdos, Carl Pomerance, JOURNAL OF NUMBER THEORY, Vol. 17, No. 1. August 1983. See section 7 for a description of an algorithm similar to Knuth's.
- [Yorgey] Generating Multiset Partitions, Brent Yorgey, The Monad.Reader, Issue 8, September 2007.
- [R557] Generating Integer Partitions, [online], Available: <http://jeromekelleher.net/generating-integer-partitions.html>
- [R558] Jerome Kelleher and Barry O'Sullivan, "Generating All Partitions: A Comparison Of Two Encodings", [online], Available: <http://arxiv.org/pdf/0909.2331v2.pdf>
- [R559] <http://stackoverflow.com/questions/6116978/python-replace-multiple-strings>
- [R22] http://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange_equation
- [R23] http://en.wikipedia.org/wiki/Mathematical_singularity
- [R24] Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg; Mathematics of computation; 51; 184; (1988); 699-706; doi:10.1090/S0025-5718-1988-0935077-0
- [R422] https://en.wikipedia.org/wiki/DFT_matrix
- [R423] http://en.wikipedia.org/wiki/Gamma_matrices
- [R424] http://en.wikipedia.org/wiki/Pauli_matrices
- [R429] http://en.wikipedia.org/wiki/Pauli_matrices
- [R447] http://en.wikipedia.org/wiki/Kronecker_delta
- [Rasch03] J. Rasch and A. C. H. Yu, 'Efficient Storage Scheme for Pre-calculated Wigner 3j, 6j and Gaunt Coefficients', SIAM J. Sci. Comput. Volume 25, Issue 4, pp. 1416-1428 (2003)
- [Liberatodebrito82] 'FORTRAN program for the integral of three spherical harmonics', A. Liberato de Brito, Comput. Phys. Commun., Volume 25, pp. 81-85 (1982)
- [Regge58] 'Symmetry Properties of Clebsch-Gordan Coefficients', T. Regge, Nuovo Cimento, Volume 10, pp. 544 (1958)
- [Edmonds74] 'Angular Momentum in Quantum Mechanics', A. R. Edmonds, Princeton University Press (1974)
- [Regge59] 'Symmetry Properties of Racah Coefficients', T. Regge, Nuovo Cimento, Volume 11, pp. 116 (1959)
- [Page52] C. H. Page, [Classes of units in the SI](#), Am. J. of Phys. 20, 1 (1952): 1.
- [Page78] C. H. Page, [Units and Dimensions in Physics](#), Am. J. of Phys. 46, 1 (1978): 78.
- [deBoer79] J. de Boer, [Group properties of quantities and units](#), Am. J. of Phys. 47, 9 (1979): 818.
- [LevyLeblond77] J.-M. Lévy-Leblond, [On the Conceptual Nature of the Physical Constants](#), La Rivista Del Nuovo Cimento 7, no. 2 (1977): 187-214.
- [NIST] [NIST reference on constants, units and uncertainties](#).
- [WikiDyadics] "Dyadics." Wikipedia, the Free Encyclopedia. Web. 05 Aug. 2011. <<http://en.wikipedia.org/wiki/Dyadics>>.

- [WikiDyadicProducts] "Dyadic Product." Wikipedia, the Free Encyclopedia. Web. 05 Aug. 2011. <http://en.wikipedia.org/wiki/Dyadic_product>.
- [Likins1973] Likins, Peter W. Elements of Engineering Mechanics. McGraw-Hill, Inc. 1973. Print.
- [Blajer1994] Blajer, Wojciech, Werner Schiehlen, and Walter Schirm. "A projective criterion to the coordinate partitioning method for multibody dynamics." Archive of Applied Mechanics 64: 86-98. Print.
- [Kane1983] Kane, Thomas R., Peter W. Likins, and David A. Levinson. Spacecraft Dynamics. New York: McGraw-Hill Book, 1983. Print.
- [Kane1985] Kane, Thomas R., and David A. Levinson. Dynamics, Theory and Applications. New York: McGraw-Hill, 1985. Print.
- [Meijaard2007] Meijaard, J.P., Jim M. Papadopoulos, Andy Ruina, and A.L. Schwab. "Linearized Dynamics Equations for the Balance and Steer of a Bicycle: a Benchmark and Review." Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences 463.2084 (2007): 1955-982. Print.
- [R430] <http://en.wikipedia.org/wiki/Commutator>
- [R431] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R432] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R433] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R434] <http://en.wikipedia.org/wiki/Commutator>
- [R435] http://en.wikipedia.org/wiki/Hermitian_adjoint
- [R436] http://en.wikipedia.org/wiki/Hermitian_transpose
- [R439] http://en.wikipedia.org/wiki/Inner_product
- [R437] http://en.wikipedia.org/wiki/Hilbert_space
- [R438] http://en.wikipedia.org/wiki/Fock_space
- [R440] http://en.wikipedia.org/wiki/Operator_%28physics%29
- [R441] <http://en.wikipedia.org/wiki/Observable>
- [R442] http://en.wikipedia.org/wiki/Outer_product
- [R443] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R444] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
- [R445] http://en.wikipedia.org/wiki/Bra-ket_notation
- [R446] http://en.wikipedia.org/wiki/Bra-ket_notation
- [R425] http://en.wikipedia.org/wiki/Ray_transfer_matrix_analysis
- [R426] http://en.wikipedia.org/wiki/Complex_beam_parameter
- [R427] http://en.wikipedia.org/wiki/Gaussian_beam
- [R428] http://en.wikipedia.org/wiki/Optical_medium
- [WikiDel] <http://en.wikipedia.org/wiki/Del>
- [R560] http://en.wikipedia.org/wiki/Dyadic_tensor
- [R561] Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill
- [Dyadics] <http://en.wikipedia.org/wiki/Dyadics>
- [DyadicProducts] http://en.wikipedia.org/wiki/Dyadic_product

[DelOperator] <http://en.wikipedia.org/wiki/Del>

PYTHON MODULE INDEX

S

- `sympy`, 776
- `sympy.assumptions`, 1076
- `sympy.assumptions.ask`, 1076
- `sympy.assumptions.assume`, 1092
- `sympy.assumptions.handlers`, 1095
- `sympy.assumptions.handlers.calculus`, 1095
- `sympy.assumptions.handlers.ntheory`, 1097
- `sympy.assumptions.handlers.order`, 1097
- `sympy.assumptions.handlers.sets`, 1098
- `sympy.assumptions.refine`, 1094
- `sympy.calculus`, 1460
- `sympy.calculus.euler`, 1460
- `sympy.calculus.finite_diff`, 1463
- `sympy.calculus.singularities`, 1461
- `sympy.categories`, 1736
- `sympy.categories.diagram_drawing`, 1745
- `sympy.codegen.algorithms`, 421
- `sympy.codegen.approximations`, 391
- `sympy.codegen.ast`, 393
- `sympy.codegen.cfunctions`, 409
- `sympy.codegen.cnodes`, 413
- `sympy.codegen.cutils`, 423
- `sympy.codegen.cxxnodes`, 414
- `sympy.codegen.fnodes`, 414
- `sympy.codegen.futils`, 423
- `sympy.codegen.pyutils`, 422
- `sympy.codegen.rewriting`, 389
- `sympy.combinatorics.generators`, 231
- `sympy.combinatorics.graycode`, 274
- `sympy.combinatorics.group_constructs`, 287
- `sympy.combinatorics.named_groups`, 279
- `sympy.combinatorics.partitions`, 202
- `sympy.combinatorics.perm_groups`, 232
- `sympy.combinatorics.permutations`, 207
- `sympy.combinatorics.polyhedron`, 261
- `sympy.combinatorics.prufer`, 264
- `sympy.combinatorics.subsets`, 268
- `sympy.combinatorics.tensor_can`, 289
- `sympy.combinatorics.testutil`, 287
- `sympy.combinatorics.util`, 281
- `sympy.core.add`, 158
- `sympy.core.assumptions`, 94
- `sympy.core.basic`, 96
- `sympy.core.cache`, 96
- `sympy.core.compatibility`, 198
- `sympy.core.containers`, 197
- `sympy.core.core`, 110
- `sympy.core.evalf`, 196
- `sympy.core.expr`, 111
- `sympy.core.exprtools`, 200
- `sympy.core.function`, 176
- `sympy.core.mod`, 161
- `sympy.core.mul`, 155
- `sympy.core.multidimensional`, 175
- `sympy.core.numbers`, 138
- `sympy.core.power`, 152
- `sympy.core.relational`, 161
- `sympy.core.singleton`, 110
- `sympy.core.symbol`, 133
- `sympy.core.sympify`, 91
- `sympy.crypto.crypto`, 336
- `sympy.diffgeom`, 1753
- `sympy.functions`, 425
- `sympy.functions.special.bessel`, 512
- `sympy.functions.special.beta_functions`, 490
- `sympy.functions.special.elliptic_integrals`, 534
- `sympy.functions.special.error_functions`, 491
- `sympy.functions.special.gamma_functions`, 483
- `sympy.functions.special.mathieu_functions`, 537
- `sympy.functions.special.polynomials`, 540
- `sympy.functions.special.singularity_functions`, 481
- `sympy.functions.special.zeta_functions`, 524

`sympy.geometry.curve`, 608
`sympy.geometry.ellipse`, 612
`sympy.geometry.entity`, 561
`sympy.geometry.line`, 579
`sympy.geometry.plane`, 650
`sympy.geometry.point`, 567
`sympy.geometry.polygon`, 627
`sympy.geometry.util`, 564
`sympy.holonomic`, 657
`sympy.holonomic.holonomic`, 665
`sympy.integrals`, 666
`sympy.integrals.intpoly`, 700
`sympy.integrals.meijerint_doc`, 684
`sympy.integrals.transforms`, 667
`sympy.liealgebras`, 703
`sympy.logic`, 717
`sympy.matrices`, 729
`sympy.matrices.expressions`, 778
`sympy.matrices.expressions.blockmatrix`, 784
`sympy.matrices.immutable`, 777
`sympy.matrices.matrices`, 729
`sympy.matrices.sparse`, 767
`sympy.ntheory.continued_fraction`, 330
`sympy.ntheory.egyptian_fraction`, 334
`sympy.ntheory.factor_`, 306
`sympy.ntheory.generate`, 298
`sympy.ntheory.modular`, 322
`sympy.ntheory.multinomial`, 324
`sympy.ntheory.partitions_`, 325
`sympy.ntheory.primetest`, 325
`sympy.ntheory.residue_ntheory`, 326
`sympy.physics`, 1469
`sympy.physics.continuum_mechanics.beam`, 1722
`sympy.physics.hep.gamma_matrices`, 1514
`sympy.physics.hydrogen`, 1469
`sympy.physics.matrices`, 1472
`sympy.physics.mechanics.body`, 1644
`sympy.physics.mechanics.kane`, 1631
`sympy.physics.mechanics.lagrange`, 1634
`sympy.physics.mechanics.linearize`, 1641
`sympy.physics.mechanics.particle`, 1621
`sympy.physics.mechanics.rigidbody`, 1623
`sympy.physics.mechanics.system`, 1637
`sympy.physics.optics.gaussopt`, 1702
`sympy.physics.optics.medium`, 1713
`sympy.physics.optics.utils`, 1715
`sympy.physics.optics.waves`, 1719
`sympy.physics.paulialgebra`, 1474
`sympy.physics.qho_ld`, 1474
`sympy.physics.quantum.anticommutator`, 1647
`sympy.physics.quantum.cartesian`, 1656
`sympy.physics.quantum.cg`, 1648
`sympy.physics.quantum.circuitplot`, 1688
`sympy.physics.quantum.commutator`, 1650
`sympy.physics.quantum.constants`, 1651
`sympy.physics.quantum.dagger`, 1652
`sympy.physics.quantum.gate`, 1689
`sympy.physics.quantum.grover`, 1694
`sympy.physics.quantum.hilbert`, 1657
`sympy.physics.quantum.innerproduct`, 1653
`sympy.physics.quantum.operator`, 1658
`sympy.physics.quantum.operatorset`, 1663
`sympy.physics.quantum.piab`, 1701
`sympy.physics.quantum.qapply`, 1665
`sympy.physics.quantum.qft`, 1696
`sympy.physics.quantum.qubit`, 1696
`sympy.physics.quantum.represent`, 1666
`sympy.physics.quantum.shor`, 1701
`sympy.physics.quantum.spin`, 1670
`sympy.physics.quantum.state`, 1680
`sympy.physics.quantum.tensorproduct`, 1654
`sympy.physics.secondquant`, 1476
`sympy.physics.sho`, 1475
`sympy.physics.units.dimensions`, 1509
`sympy.physics.units.prefixes`, 1512
`sympy.physics.units.quantities`, 1513
`sympy.physics.units.unitsystem`, 1512
`sympy.physics.units.util`, 1513
`sympy.physics.vector.functions`, 1569
`sympy.physics.vector.point`, 1564
`sympy.physics.wigner`, 1497
`sympy.plotting.plot`, 1059
`sympy.plotting.pygletplot`, 1072
`sympy.printing.ccode`, 1024
`sympy.printing.codeprinter`, 1053
`sympy.printing.conventions`, 1053
`sympy.printing.cxxcode`, 1027
`sympy.printing.fcode`, 1030
`sympy.printing.gtk`, 1044
`sympy.printing.jscode`, 1035
`sympy.printing.julia`, 1037
`sympy.printing.lambdarepr`, 1045
`sympy.printing.latex`, 1045
`sympy.printing.mathematica`, 1034
`sympy.printing.mathml`, 1047
`sympy.printing.octave`, 1039
`sympy.printing.precedence`, 1054
`sympy.printing.pretty.pretty`, 1023
`sympy.printing.pretty.pretty_symbology`, 1055
`sympy.printing.pretty.stringpict`, 1056
`sympy.printing.preview`, 1052
`sympy.printing.printer`, 1019

- [sympy.printing.pycode](#), 1048
- [sympy.printing.python](#), 1049
- [sympy.printing.rcode](#), 1028
- [sympy.printing.repr](#), 1049
- [sympy.printing.rust](#), 1042
- [sympy.printing.str](#), 1050
- [sympy.printing.theanocode](#), 1044
- [sympy.printing.tree](#), 1050
- [sympy.sets.fancysets](#), 1146
- [sympy.sets.sets](#), 1132
- [sympy.simplify.combsimp](#), 1173
- [sympy.simplify.cse_main](#), 1174
- [sympy.simplify.epathtools](#), 1177
- [sympy.simplify.hyperexpand](#), 1176
- [sympy.simplify.hyperexpand_doc](#), 1187
- [sympy.simplify.powsimp](#), 1170
- [sympy.simplify.radsimp](#), 1162
- [sympy.simplify.ratsimp](#), 1169
- [sympy.simplify.simplify](#), 77
- [sympy.simplify.sqrtdenest](#), 1173
- [sympy.simplify.traversaltools](#), 1177
- [sympy.simplify.trigsimp](#), 1169
- [sympy.solvers](#), 1295
- [sympy.solvers.inequalities](#), 1339
- [sympy.solvers.ode](#), 1283
- [sympy.solvers.pde](#), 1295
- [sympy.solvers.recurr](#), 1310
- [sympy.solvers.solve](#), 1342
- [sympy.stats](#), 1189
- [sympy.stats.crv](#), 1229
- [sympy.stats.crv_types](#), 1229
- [sympy.stats.frv](#), 1229
- [sympy.stats.frv_types](#), 1229
- [sympy.stats.rv](#), 1228
- [sympy.tensor](#), 1360
- [sympy.tensor.array](#), 1360
- [sympy.tensor.index_methods](#), 1372
- [sympy.tensor.indexed](#), 1366
- [sympy.tensor.tensor](#), 1375
- [sympy.utilities](#), 1390
- [sympy.utilities.autowrap](#), 1391
- [sympy.utilities.codegen](#), 1397
- [sympy.utilities.decorator](#), 1409
- [sympy.utilities.enumerative](#), 1410
- [sympy.utilities.iterables](#), 1417
- [sympy.utilities.lambdify](#), 1437
- [sympy.utilities.memoization](#), 1441
- [sympy.utilities.misc](#), 1441
- [sympy.utilities.pkgdata](#), 1444
- [sympy.utilities.pytest](#), 1445
- [sympy.utilities.randtest](#), 1446
- [sympy.utilities.runttests](#), 1447
- [sympy.utilities.source](#), 1454
- [sympy.utilities.timeutils](#), 1454

Symbols

<code>_TensorManager</code>	(class in <code>sympy.tensor.tensor</code>), 1375	<code>_linear_2eq_order2_type11()</code>	(in <code>sympy.solvers.ode</code>), 1276	module
<code>__init__()</code>	(<code>sympy.vector.coordsysrect.CoordSys3D</code> method), 1786	<code>_linear_2eq_order2_type2()</code>	(in <code>sympy.solvers.ode</code>), 1270	module
<code>__init__()</code>	(<code>sympy.vector.orienters.AxisOrienter</code> method), 1798	<code>_linear_2eq_order2_type3()</code>	(in <code>sympy.solvers.ode</code>), 1271	module
<code>__init__()</code>	(<code>sympy.vector.orienters.BodyOrienter</code> method), 1799	<code>_linear_2eq_order2_type4()</code>	(in <code>sympy.solvers.ode</code>), 1271	module
<code>__init__()</code>	(<code>sympy.vector.orienters.QuaternionOrienter</code> method), 1801	<code>_linear_2eq_order2_type5()</code>	(in <code>sympy.solvers.ode</code>), 1272	module
<code>__init__()</code>	(<code>sympy.vector.orienters.SpaceOrienter</code> method), 1800	<code>_linear_2eq_order2_type6()</code>	(in <code>sympy.solvers.ode</code>), 1273	module
<code>_base_ordering()</code>	(in <code>sympy.combinatorics.util</code>), 281	<code>_linear_2eq_order2_type7()</code>	(in <code>sympy.solvers.ode</code>), 1273	module
<code>_check_cycles_alt_sym()</code>	(in <code>sympy.combinatorics.util</code>), 282	<code>_linear_2eq_order2_type8()</code>	(in <code>sympy.solvers.ode</code>), 1274	module
<code>_cmp_perm_lists()</code>	(in <code>sympy.combinatorics.testutil</code>), 287	<code>_linear_2eq_order2_type9()</code>	(in <code>sympy.solvers.ode</code>), 1275	module
<code>_distribute_gens_by_base()</code>	(in <code>sympy.combinatorics.util</code>), 282	<code>_linear_3eq_order1_type1()</code>	(in <code>sympy.solvers.ode</code>), 1276	module
<code>_handle_precomputed_bsgs()</code>	(in <code>sympy.combinatorics.util</code>), 283	<code>_linear_3eq_order1_type2()</code>	(in <code>sympy.solvers.ode</code>), 1277	module
<code>_linear_2eq_order1_type1()</code>	(in <code>sympy.solvers.ode</code>), 1266	<code>_linear_3eq_order1_type3()</code>	(in <code>sympy.solvers.ode</code>), 1277	module
<code>_linear_2eq_order1_type2()</code>	(in <code>sympy.solvers.ode</code>), 1267	<code>_linear_3eq_order1_type4()</code>	(in <code>sympy.solvers.ode</code>), 1278	module
<code>_linear_2eq_order1_type3()</code>	(in <code>sympy.solvers.ode</code>), 1267	<code>_linear_neq_order1_type1()</code>	(in <code>sympy.solvers.ode</code>), 1278	module
<code>_linear_2eq_order1_type4()</code>	(in <code>sympy.solvers.ode</code>), 1268	<code>_naive_list_centralizer()</code>	(in <code>sympy.combinatorics.testutil</code>), 288	module
<code>_linear_2eq_order1_type5()</code>	(in <code>sympy.solvers.ode</code>), 1268	<code>_nonlinear_2eq_order1_type1()</code>	(in <code>sympy.solvers.ode</code>), 1279	module
<code>_linear_2eq_order1_type6()</code>	(in <code>sympy.solvers.ode</code>), 1268	<code>_nonlinear_2eq_order1_type2()</code>	(in <code>sympy.solvers.ode</code>), 1280	module
<code>_linear_2eq_order1_type7()</code>	(in <code>sympy.solvers.ode</code>), 1269	<code>_nonlinear_2eq_order1_type3()</code>	(in <code>sympy.solvers.ode</code>), 1280	module
<code>_linear_2eq_order2_type1()</code>	(in <code>sympy.solvers.ode</code>), 1269	<code>_nonlinear_2eq_order1_type4()</code>	(in <code>sympy.solvers.ode</code>), 1280	module
<code>_linear_2eq_order2_type10()</code>	(in <code>sympy.solvers.ode</code>), 1275	<code>_nonlinear_2eq_order1_type5()</code>	(in <code>sympy.solvers.ode</code>), 1281	module
		<code>_nonlinear_3eq_order1_type1()</code>	(in <code>sympy.solvers.ode</code>), 1281	module

`_nonlinear_3eq_order1_type2()` (in module `sympy.solvers.ode`), 1282
`_nonlinear_3eq_order1_type3()` (in module `sympy.solvers.ode`), 1282
`_nonlinear_3eq_order1_type4()` (in module `sympy.solvers.ode`), 1283
`_nonlinear_3eq_order1_type5()` (in module `sympy.solvers.ode`), 1283
`_orbits_transversals_from_bsgs()` (in module `sympy.combinatorics.util`), 284
`_print()` (`sympy.printing.printer.Printer` method), 1022
`_remove_gens()` (in module `sympy.combinatorics.util`), 284
`_strip()` (in module `sympy.combinatorics.util`), 285
`_strong_gens_from_distr()` (in module `sympy.combinatorics.util`), 286
`_verify_bsgs()` (in module `sympy.combinatorics.testutil`), 288
`_verify_centralizer()` (in module `sympy.combinatorics.testutil`), 288
`_verify_normal_closure()` (in module `sympy.combinatorics.testutil`), 288

A

`A` (`sympy.physics.optics.gaussopt.RayTransferMatrix` attribute), 1703
`a` (`sympy.physics.quantum.shor.CMod` attribute), 1701
`a1pt_theory()` (`sympy.physics.vector.point.Point` method), 1564
`a2idx()` (in module `sympy.matrices.matrices`), 762
`a2pt_theory()` (`sympy.physics.vector.point.Point` method), 1565
`a_interval` (`sympy.sets.fancysets.ComplexRegion` attribute), 1152
`abbrev` (`sympy.physics.units.quantities.Quantity` attribute), 1513
`AbelianGroup()` (in module `sympy.combinatorics.named_groups`), 281
`above()` (`sympy.printing.pretty.stringpict.stringpic` method), 1056
`Abs` (class in `sympy.functions.elementary.complexes`), 429
`abs()` (`sympy.polys.domains.domain.Domain` method), 905
`abs()` (`sympy.polys.polyclasses.DMP` method), 918
`abs()` (`sympy.polys.polytools.Poly` method), 823
`acc()` (`sympy.physics.vector.point.Point` method), 1565
`accepted_latex_functions` (in module `sympy.printing.latex`), 1045
`acos` (class in `sympy.functions.elementary.trigonometric`), 438
`acosh` (class in `sympy.functions.elementary.hyperbolic`), 445
`acot` (class in `sympy.functions.elementary.trigonometric`), 439
`acoth` (class in `sympy.functions.elementary.hyperbolic`), 445
`acsc` (class in `sympy.functions.elementary.trigonometric`), 441
`acsch` (class in `sympy.functions.elementary.hyperbolic`), 446
`Add` (class in `sympy.core.add`), 158
`add()` (`sympy.assumptions.assume.AssumptionsContext` method), 1093
`Add()` (`sympy.assumptions.handlers.calculus.AskFiniteHand` static method), 1096
`Add()` (`sympy.assumptions.handlers.order.AskNegativeHand` static method), 1098
`Add()` (`sympy.assumptions.handlers.sets.AskAntiHermitianH` static method), 1098
`Add()` (`sympy.assumptions.handlers.sets.AskHermitianHand` static method), 1098
`Add()` (`sympy.assumptions.handlers.sets.AskImaginaryHand` static method), 1099
`Add()` (`sympy.assumptions.handlers.sets.AskIntegerHandler` static method), 1099
`Add()` (`sympy.assumptions.handlers.sets.AskRationalHandle` static method), 1099
`Add()` (`sympy.assumptions.handlers.sets.AskRealHandler` static method), 1099
`add()` (`sympy.matrices.matrices.MatrixBase` method), 743
`add()` (`sympy.polys.domains.domain.Domain` method), 905
`add()` (`sympy.polys.polyclasses.DMF` method), 924
`add()` (`sympy.polys.polyclasses.DMP` method), 918
`add()` (`sympy.polys.polytools.Poly` method), 824
`add()` (`sympy.polys.rings.PolyRing` method), 973
`add_as_roots()` (`sympy.liealgebras.root_system.RootSystem` method), 703
`add_gens()` (`sympy.polys.rings.PolyRing` method), 973
`add_ground()` (`sympy.polys.polyclasses.DMP` method), 918

- [add_ground\(\)](#) (sympy.polys.polytools.Poly method), [824](#)
[add_simple_roots\(\)](#) (sympy.liealgebras.root_system.RootSystem.degen.fnodes), [417](#)
[method](#), [704](#)
[affine_rank\(\)](#) (sympy.geometry.point.Point static method), [568](#)
[airyai](#) (class in sympy.functions.special.bessel), [517](#)
[airyaiprime](#) (class in sympy.functions.special.bessel), [520](#)
[AiryBase](#) (class in sympy.functions.special.bessel), [517](#)
[airybi](#) (class in sympy.functions.special.bessel), [519](#)
[airybiprime](#) (class in sympy.functions.special.bessel), [521](#)
[alg_con](#) (sympy.physics.mechanics.system.SymbolicSystem attribute), [1640](#)
[algebraic](#), [95](#)
[algebraic](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1076](#)
[algebraic_field\(\)](#) (sympy.polys.domains.AlgebraicField method), [914](#)
[algebraic_field\(\)](#) (sympy.polys.domains.domain.Domain method), [905](#)
[algebraic_field\(\)](#) (sympy.polys.domains.IntegerRing method), [912](#)
[algebraic_field\(\)](#) (sympy.polys.domains.RationalField method), [913](#)
[AlgebraicField](#) (class in sympy.polys.domains), [914](#)
[AlgebraicNumber](#) (class in sympy.polys.numberfields), [867](#)
[alignof\(\)](#) (in module sympy.codegen.cnodes), [413](#)
[all_coeffs\(\)](#) (sympy.polys.polyclasses.DMP method), [918](#)
[all_coeffs\(\)](#) (sympy.polys.polytools.Poly method), [824](#)
[all_monoms\(\)](#) (sympy.polys.polyclasses.DMP method), [918](#)
[all_monoms\(\)](#) (sympy.polys.polytools.Poly method), [824](#)
[all_roots\(\)](#) (sympy.liealgebras.root_system.RootSystem method), [704](#)
[all_roots\(\)](#) (sympy.polys.polytools.Poly method), [825](#)
[all_terms\(\)](#) (sympy.polys.polyclasses.DMP method), [919](#)
[all_terms\(\)](#) (sympy.polys.polytools.Poly method), [825](#)
[allhints](#) (in module sympy.solvers.ode), [1238](#)
[allocated\(\)](#) (in module sympy.codegen.fnodes), [417](#)
[almosteq\(\)](#) (sympy.polys.domains.domain.Domain method), [905](#)
[almosteq\(\)](#) (sympy.polys.domains.RealField method), [916](#)
[almosteq\(\)](#) (sympy.polys.rings.PolyElement method), [974](#)
[alternating\(\)](#) (sympy.combinatorics.generators method), [232](#)
[AlternatingGroup\(\)](#) (in module sympy.combinatorics.named_groups), [280](#)
[altitudes](#) (sympy.geometry.polygon.Triangle attribute), [642](#)
[ambient_dimension](#) (sympy.geometry.entity.GeometryEntity attribute), [561](#)
[ambient_dimension](#) (sympy.geometry.line.LinearEntity attribute), [580](#)
[ambient_dimension](#) (sympy.geometry.point.Point attribute), [568](#)
[amplitude](#) (sympy.physics.optics.waves.TWave attribute), [1720](#)
[amplitude](#) (sympy.functions.special.hyper.meijerg attribute), [534](#)
[And](#) (class in sympy.logic.boolalg), [721](#)
[ang_acc_in\(\)](#) (sympy.physics.vector.frame.ReferenceFrame method), [1552](#)
[ang_vel_in\(\)](#) (sympy.physics.vector.frame.ReferenceFrame method), [1552](#)
[angle](#) (sympy.physics.optics.gaussopt.GeometricRay attribute), [1708](#)
[angle_between\(\)](#) (sympy.geometry.line.LinearEntity method), [580](#)
[angle_between\(\)](#) (sympy.geometry.plane.Plane method), [650](#)
[angles](#) (sympy.geometry.polygon.Polygon attribute), [628](#)
[angles](#) (sympy.geometry.polygon.RegularPolygon attribute), [635](#)
[angular_momentum\(\)](#) (in module sympy.physics.mechanics.functions), [1628](#)
[angular_momentum\(\)](#) (sympy.physics.mechanics.particle.Particle method), [1621](#)
[angular_momentum\(\)](#) (sympy.physics.mechanics.rigidbody.RigidBody method), [1624](#)
[angular_velocity](#) (sympy.physics.optics.waves.TWave attribute), [1720](#)
[AnnihilateBoson](#) (class in sympy.physics.secondquant), [1481](#)

AnnihilateFermion (class in sympy.physics.secondquant), 1482

annotated() (in module sympy.printing.pretty.pretty_symbology), 1056

ANP (class in sympy.polys.polyclasses), 924

AntiCommutator (class in sympy.physics.quantum.anticommutator), 1647

antidivisor_count() (in module sympy.ntheory.factor_), 317

antidivisors() (in module sympy.ntheory.factor_), 316

antihermitian, 96

antihermitian (sympy.assumptions.ask.AssumptionRef attribute), 1076

AntiSymmetricTensor (class in sympy.physics.secondquant), 1494

aother (sympy.functions.special.hyper.meijerg attribute), 534

ap (sympy.functions.special.hyper.hyper attribute), 531

ap (sympy.functions.special.hyper.meijerg attribute), 534

apart() (in module sympy.polys.partfrac), 876

apart() (sympy.core.expr.Expr method), 111

apart_list() (in module sympy.polys.partfrac), 877

apoapsis (sympy.geometry.ellipse.Ellipse attribute), 613

apothem (sympy.geometry.polygon.RegularPolygon attribute), 635

append() (sympy.plotting.plot.Plot method), 1060

applied_loads (sympy.physics.continuum_mechanics.beam.Beam attribute), 1723

AppliedPredicate (class in sympy.assumptions.assume), 1092

apply() (sympy.printing.pretty.stringpict.prettyForm static method), 1057

apply() (sympy.simplify.epathtools.EPath method), 1177

apply_finite_diff() (in module sympy.calculus.finite_diff), 1464

apply_force() (sympy.physics.mechanics.body.Body method), 1646

apply_grover() (in module sympy.physics.quantum.grover), 1695

apply_load() (sympy.physics.continuum_mechanics.beam.Beam method), 1724

apply_operator() (sympy.physics.secondquant.AnnihilateFermion method), 1481

apply_operator() (sympy.physics.secondquant.AnnihilateFermion method), 1482

apply_operator() (sympy.physics.secondquant.CreateBoson method), 1482

apply_operator() (sympy.physics.secondquant.CreateFermion method), 1483

apply_operators() (in module sympy.physics.secondquant), 1486

apply_support() (sympy.physics.continuum_mechanics.beam.Beam method), 1725

apply_torque() (sympy.physics.mechanics.body.Body method), 1646

applyfunc() (sympy.matrices.sparse.SparseMatrix method), 768

applyfunc() (sympy.physics.vector.dyadic.Dyadic method), 1561

applyfunc() (sympy.physics.vector.vector.Vector method), 1557

approximation() (sympy.core.numbers.NumberSymbol method), 143

arbitrary_point() (sympy.geometry.curve.Curve method), 608

arbitrary_point() (sympy.geometry.ellipse.Ellipse method), 613

arbitrary_point() (sympy.geometry.line.LinearEntity method), 581

arbitrary_point() (sympy.geometry.plane.Plane method), 650

arbitrary_point() (sympy.geometry.polygon.Polygon method), 628

asin() (in module sympy.stats), 1193

are_collinear() (sympy.geometry.point.Point3D static method), 576

are_concurrent() (sympy.geometry.line.LinearEntity static method), 581

are_concurrent() (sympy.geometry.plane.Plane static method), 651

are_coplanar() (sympy.geometry.point.Point3D static method), 568

are_similar() (in module sympy.geometry.util), 565

area (sympy.geometry.ellipse.Ellipse attribute), 613

area (sympy.geometry.polygon.Polygon attribute), 629

area (sympy.geometry.polygon.RegularPolygon attribute), 635

arg (class in sympy.functions.elementary.complexes), 430

args (sympy.assumptions.assume.AppliedPredicate attribute), 1092

args (sympy.core.basic.Basic attribute), 97

args (sympy.geometry.polygon.RegularPolygon attribute), 636

[args](#) (`sympy.polys.polytools.Poly` attribute), [825](#)
[args_cnc\(\)](#) (`sympy.core.expr.Expr` method), [111](#)
[Argument](#) (class in `sympy.utilities.codegen`), [1399](#)
[argument](#) (`sympy.functions.special.bessel.BesselBase` attribute), [512](#)
[argument](#) (`sympy.functions.special.hyper.hyperas` attribute), [531](#)
[argument](#) (`sympy.functions.special.hyper.meijer` attribute), [534](#)
[array\(\)](#) (in module `sympy.codegen.fnodes`), [417](#)
[array_form](#) (`sympy.combinatorics.permutations.Permutation` attribute), [212](#)
[array_form](#) (`sympy.combinatorics.polyhedron.Polyhedron` attribute), [261](#)
[ArrayConstructor](#) (class in `sympy.codegen.fnodes`), [414](#)
[ArrowStringDescription](#) (class in `sympy.categories.diagram_drawing`), [1748](#)
[as_base_exp\(\)](#) (`sympy.core.function.Function` method), [185](#)
[as_base_exp\(\)](#) (`sympy.core.power.Pow` method), [153](#)
[as_coeff_Add\(\)](#) (`sympy.core.add.Add` method), [158](#)
[as_coeff_add\(\)](#) (`sympy.core.add.Add` method), [158](#)
[as_coeff_Add\(\)](#) (`sympy.core.expr.Expr` method), [112](#)
[as_coeff_add\(\)](#) (`sympy.core.expr.Expr` method), [112](#)
[as_coeff_Add\(\)](#) (`sympy.core.numbers.Number` method), [138](#)
[as_coeff_Add\(\)](#) (`sympy.core.numbers.Rational` method), [142](#)
[as_coeff_exponent\(\)](#) (`sympy.core.expr.Expr` method), [112](#)
[as_coeff_Mul\(\)](#) (`sympy.core.expr.Expr` method), [112](#)
[as_coeff_mul\(\)](#) (`sympy.core.expr.Expr` method), [112](#)
[as_coeff_Mul\(\)](#) (`sympy.core.mul.Mul` method), [155](#)
[as_coeff_Mul\(\)](#) (`sympy.core.numbers.Number` method), [138](#)
[as_coeff_Mul\(\)](#) (`sympy.core.numbers.Rational` method), [142](#)
[as_coeff_Mul\(\)](#) (`sympy.core.numbers.Zero` method), [145](#)
[as_coeff_Mul\(\)](#) (`sympy.matrices.expressions.MatrixExpr` method), [779](#)
[as_coefficient\(\)](#) (`sympy.core.expr.Expr` method), [113](#)
[as_coefficients_dict\(\)](#) (`sympy.core.add.Add` method), [158](#)
[as_coefficients_dict\(\)](#) (`sympy.core.expr.Expr` method), [114](#)
[as_coefficients_dict\(\)](#) (`sympy.core.mul.Mul` method), [155](#)
[as_content_primitive\(\)](#) (`sympy.core.add.Add` method), [159](#)
[as_content_primitive\(\)](#) (`sympy.core.basic.Basic` method), [159](#)
[as_content_primitive\(\)](#) (`sympy.core.expr.Expr` method), [114](#)
[as_content_primitive\(\)](#) (`sympy.core.mul.Mul` method), [155](#)
[as_content_primitive\(\)](#) (`sympy.core.numbers.Rational` method), [142](#)
[as_content_primitive\(\)](#) (`sympy.core.power.Pow` method), [154](#)
[as_Declaration\(\)](#) (`sympy.codegen.ast.Variable` method), [407](#)
[as_dict\(\)](#) (`sympy.combinatorics.partitions.IntegerPartition` method), [204](#)
[as_dict\(\)](#) (`sympy.polys.polytools.Poly` method), [826](#)
[as_dummy\(\)](#) (`sympy.core.symbol.Symbol` method), [133](#)
[as_explicit\(\)](#) (`sympy.matrices.expressions.MatrixExpr` method), [779](#)
[as_expr\(\)](#) (`sympy.core.expr.Expr` method), [115](#)
[as_expr\(\)](#) (`sympy.polys.numberfields.AlgebraicNumber` method), [867](#)
[as_expr\(\)](#) (`sympy.polys.polytools.Poly` method), [826](#)
[as_ferrers\(\)](#) (`sympy.combinatorics.partitions.IntegerPartition` method), [204](#)
[as_finite_diff\(\)](#) (in module `sympy.calculus.finite_diff`), [1465](#)
[as_finite_difference\(\)](#) (`sympy.core.function.Derivative` method), [180](#)
[as_immutable\(\)](#) (`sympy.matrices.sparse.SparseMatrix` method), [768](#)
[as_independent\(\)](#) (`sympy.core.expr.Expr` method), [115](#)
[as_int\(\)](#) (in module `sympy.core.compatibility`), [200](#)

`as_leading_term()` (sympy.core.expr.Expr method), 118

`as_list()` (sympy.polys.polytools.Poly method), 826

`as_mutable()` (sympy.matrices.expressions.MatrixExpr method), 779

`as_mutable()` (sympy.matrices.sparse.SparseMatrix method), 768

`as_numer_denom()` (sympy.core.expr.Expr method), 118

`as_ordered_factors()` (sympy.core.expr.Expr method), 118

`as_ordered_factors()` (sympy.core.mul.Mul method), 156

`as_ordered_terms()` (sympy.core.expr.Expr method), 118

`as_poly()` (sympy.core.basic.Basic method), 97

`as_poly()` (sympy.polys.numberfields.AlgebraicNumber method), 867

`as_powers_dict()` (sympy.core.expr.Expr method), 118

`as_real_imag()` (sympy.core.add.Add method), 159

`as_real_imag()` (sympy.core.expr.Expr method), 119

`as_real_imag()` (sympy.functions.elementary.complexes.IntegerComplex method), 428

`as_real_imag()` (sympy.functions.elementary.complexes.SymbolicComplex method), 427

`as_relational()` (sympy.sets.sets.FiniteSet method), 1141

`as_relational()` (sympy.sets.sets.Intersection method), 1143

`as_relational()` (sympy.sets.sets.Interval method), 1139

`as_relational()` (sympy.sets.sets.Union method), 1142

`as_sum()` (sympy.integrals.Integral method), 690

`as_terms()` (sympy.core.expr.Expr method), 119

`as_two_terms()` (sympy.core.add.Add method), 159

`as_two_terms()` (sympy.core.mul.Mul method), 156

`ascents()` (sympy.combinatorics.permutations.Permutation method), 213

`asec` (class in sympy.functions.elementary.trigonometric), 440

`asech` (class in sympy.functions.elementary.hyperbolic), 446

`asin` (class in sympy.functions.elementary.trigonometric), 437

`asinh` (class in sympy.functions.elementary.hyperbolic), 445

`ask()` (in module sympy.assumptions.ask), 1091

`ask_full_inference()` (in module sympy.assumptions.ask), 1091

`AskAlgebraicHandler` (class in sympy.assumptions.handlers.sets), 1098

`AskAntiHermitianHandler` (class in sympy.assumptions.handlers.sets), 1098

`AskComplexHandler` (class in sympy.assumptions.handlers.sets), 1098

`AskExtendedRealHandler` (class in sympy.assumptions.handlers.sets), 1098

`AskFiniteHandler` (class in sympy.assumptions.handlers.calculus), 1095

`AskHermitianHandler` (class in sympy.assumptions.handlers.sets), 1098

`AskImaginaryHandler` (class in sympy.assumptions.handlers.sets), 1099

`AskIntegerHandler` (class in sympy.assumptions.handlers.sets), 1099

`AskNegativeHandler` (class in sympy.assumptions.handlers.order), 1097

`AskNonZeroHandler` (class in sympy.assumptions.handlers.order), 1098

`AskOddHandler` (class in sympy.assumptions.handlers.ntheory), 1097

`AskPositiveHandler` (class in sympy.assumptions.handlers.order), 1098

`AskPrimeHandler` (class in sympy.assumptions.handlers.ntheory), 1097

`AskRationalHandler` (class in sympy.assumptions.handlers.sets), 1099

`AskRealHandler` (class in sympy.assumptions.handlers.sets), 1099

`assemble_partfrac_list()` (in module sympy.polys.partfrac), 878

`Assignment` (class in sympy.codegen.ast), 395

- Assignment (class in `sympy.printing.codeprinter`), 1053
- AssignmentError, 1054
- assoc_laguerre (class in `sympy.functions.special.polynomials`), 549
- assoc_legendre (class in `sympy.functions.special.polynomials`), 546
- assoc_recurrence_memo() (in module `sympy.utilities.memoization`), 1441
- assuming() (in module `sympy.assumptions.assume`), 1093
- AssumptionKeys (class in `sympy.assumptions.ask`), 1076
- assumptions0 (`sympy.core.basic.Basic` attribute), 98
- AssumptionsContext (class in `sympy.assumptions.assume`), 1092
- atan (class in `sympy.functions.elementary.trigonometric`), 439
- atan2 (class in `sympy.functions.elementary.trigonometric`), 441
- atanh (class in `sympy.functions.elementary.hyperbolic`), 445
- Atom (class in `sympy.core.basic`), 110
- AtomicExpr (class in `sympy.core.expr`), 133
- atoms() (`sympy.combinatorics.permutations.Permutation` method), 213
- atoms() (`sympy.core.basic.Basic` method), 98
- atoms_table (in module `sympy.printing.pretty.pretty_symbology`), 1055
- attr_params() (`sympy.codegen.ast.Node` method), 402
- Attribute (class in `sympy.codegen.ast`), 395
- aug_assign() (in module `sympy.codegen.ast`), 409
- AugmentedAssignment (class in `sympy.codegen.ast`), 396
- auto_number() (in module `sympy.parsing.sympy_parser`), 1458
- auto_symbol() (in module `sympy.parsing.sympy_parser`), 1458
- autowrap() (in module `sympy.utilities.autowrap`), 1393
- auxiliary_eqs (`sympy.physics.mechanics.kane.KaneMethod` attribute), 1633
- AxisOrienter (class in `sympy.vector.orienters`), 1798
- AZ() (in module `sympy.crypto.crypto`), 336
- ## B
- B (in module `sympy.physics.secondquant`), 1486
- B (`sympy.physics.optics.gaussopt.RayTransferMatrix` attribute), 1703
- b_interval (`sympy.sets.fancysets.ComplexRegion` attribute), 1152
- base (`sympy.combinatorics.perm_groups.PermutationGroup` attribute), 234
- base (`sympy.tensor.indexed.Indexed` attribute), 1369
- base_oneform() (`sympy.diffgeom.CoordSystem` method), 1755
- base_oneforms() (`sympy.diffgeom.CoordSystem` method), 1755
- base_solution_linear() (in module `sympy.solvers.diophantine`), 1322
- base_vector() (`sympy.diffgeom.CoordSystem` method), 1756
- base_vectors() (`sympy.diffgeom.CoordSystem` method), 1756
- BaseCovDerivativeOp (class in `sympy.diffgeom`), 1762
- BasePolynomialError (class in `sympy.polys.polyerrors`), 999
- BaseScalarField (class in `sympy.diffgeom`), 1757
- BaseSeries (class in `sympy.plotting.plot`), 1071
- baseswap() (`sympy.combinatorics.perm_groups.Permutation` method), 234
- BaseVectorField (class in `sympy.diffgeom`), 1758
- Basic (class in `sympy.core.basic`), 96
- basic_orbits (`sympy.combinatorics.perm_groups.Permutation` attribute), 235
- basic_root() (`sympy.liealgebras.type_a.TypeA` method), 705
- basic_root() (`sympy.liealgebras.type_b.TypeB` method), 706
- basic_root() (`sympy.liealgebras.type_c.TypeC` method), 708
- basic_root() (`sympy.liealgebras.type_d.TypeD` method), 709
- basic_root() (`sympy.liealgebras.type_e.TypeE` method), 711
- basic_root() (`sympy.liealgebras.type_f.TypeF` method), 712
- basic_stabilizers (`sympy.combinatorics.perm_groups.PermutationGroup` attribute), 236
- basic_transversals (`sympy.combinatorics.perm_groups.PermutationGroup` attribute), 236
- basis() (`sympy.liealgebras.type_a.TypeA` method), 705

basis() (sympy.liealgebras.type_b.TypeB method), 707

basis() (sympy.liealgebras.type_c.TypeC method), 708

basis() (sympy.liealgebras.type_d.TypeD method), 709

basis() (sympy.liealgebras.type_e.TypeE method), 711

basis() (sympy.liealgebras.type_f.TypeF method), 712

basis() (sympy.liealgebras.type_g.TypeG method), 713

basis() (sympy.polys.agca.modules.FreeModule method), 888

BBra (in module sympy.physics.secondquant), 1485

Bd (in module sympy.physics.secondquant), 1486

Beam (class in sympy.physics.continuum_mechanics.bifid6_square), 1722

BeamParameter (class in sympy.physics.optics.gaussopt), 1708

bell (class in sympy.functions.combinatorial.numbers), 459

below() (sympy.printing.pretty.stringpict.stringPict method), 1056

bending_moment() (sympy.physics.continuum_mechanics.bifid6_square), 1725

Benini() (in module sympy.stats), 1194

bernoulli (class in sympy.functions.combinatorial.numbers), 460

Bernoulli() (in module sympy.stats), 1190

BesselBase (class in sympy.functions.special.bessel), 512

besseli (class in sympy.functions.special.bessel), 513

besselj (class in sympy.functions.special.bessel), 512

besselk (class in sympy.functions.special.bessel), 514

besselsimp() (in module sympy.simplify.simplify), 1158

bessely (class in sympy.functions.special.bessel), 513

beta (class in sympy.functions.special.beta_functions), 490

Beta() (in module sympy.stats), 1195

BetaPrime() (in module sympy.stats), 1196

bifid5_square() (in module sympy.crypto.crypto), 346, 348

bifid6_square() (in module sympy.crypto.crypto), 349

bin_to_gray() (sympy.combinatorics.graycode method), 278

binary_function() (in module sympy.utilities.autowrap), 1394

binary_partitions() (in module sympy.utilities.iterables), 1417

bind_C() (in module sympy.codegen.fnodes), 418

binomial (class in sympy.functions.combinatorial.factorials), 461

Binomial() (in module sympy.stats), 1191

binomial_coefficients() (in module sympy.ntheory.multinomial), 324

binomial_coefficients_list() (in module sympy.ntheory.multinomial), 324

bisect() (sympy.geometry.polygon.Triangle method), 643

bitlist_from_subset() (sympy.combinatorics.subsets.Subset class method), 268

BKets (in module sympy.physics.secondquant), 1486

Block_collapse() (in module sympy.matrices.expressions.blockmatrix), 85

BlockDiagMatrix (class in sympy.matrices.expressions.blockmatrix), 784

BlockMatrix (class in sympy.matrices.expressions.blockmatrix), 784

bm (sympy.functions.special.hyper.meijerg attribute), 534

bodies (sympy.physics.mechanics.system.SymbolicSystem attribute), 1640

Body (class in sympy.physics.mechanics.body), 1644

BodyOrienter (class in sympy.vector.orienters), 1799

bool_map() (in module sympy.logic.boolalg), 727

BooleanFalse (class in sympy.logic.boolalg), 720

BooleanTrue (class in sympy.logic.boolalg), 719

BosonicBasis (class in sympy.physics.secondquant), 1486

bother (sympy.functions.special.hyper.meijerg attribute), 534

boundary (sympy.sets.sets.Set attribute), 1132

boundary_conditions (sympy.physics.continuum_mechanics.beam.Beam attribute), 1510
 attribute), 1726
 bounded (sympy.assumptions.ask.AssumptionKeys attribute), 1076
 bounds (sympy.geometry.ellipse.Ellipse attribute), 614
 bounds (sympy.geometry.entity.GeometryEntity attribute), 561
 bounds (sympy.geometry.line.LinearEntity2D attribute), 598
 bounds (sympy.geometry.point.Point2D attribute), 574
 bounds (sympy.geometry.polygon.Polygon attribute), 630
 bq (sympy.functions.special.hyper.hyper attribute), 531
 bq (sympy.functions.special.hyper.meijerg attribute), 534
 Bra (class in sympy.physics.quantum.state), 1682
 bra (sympy.physics.quantum.operator.OuterProduct attribute), 1661
 bra (sympy.physics.secondquant.InnerProduct attribute), 1486
 BraBase (class in sympy.physics.quantum.state), 1681
 bracelets() (in module sympy.utilities.iterables), 1417
 BreakToken (class in sympy.codegen.ast), 396
 brewster_angle() (in module sympy.physics.optics.utils), 1717
 bsgs_direct_product() (in module sympy.combinatorics.tensor_can), 293
 bspline_basis() (in module sympy.functions.special.bsplines), 523
 bspline_basis_set() (in module sympy.functions.special.bsplines), 524
 build_expression_tree() (in module sympy.series.gruntz), 1108
C
 C (sympy.physics.optics.gaussopt.RayTransferMatrix attribute), 1703
 C89CodePrinter (class in sympy.printing.ccode), 1024
 C99CodePrinter (class in sympy.printing.ccode), 1024
 cacheit() (in module sympy.core.cache), 96
 calculate_series() (in module sympy.series.gruntz), 1108
 can_transf_matrix (sympy.physics.units.dimensions.Dimension attribute), 1510
 canberra_distance() (sympy.geometry.point.Point method), 568
 cancel() (in module sympy.polys.polytools), 819
 cancel() (sympy.core.expr.Expr method), 119
 cancel() (sympy.polys.polyclasses.DMF method), 924
 cancel() (sympy.polys.polyclasses.DMP method), 919
 cancel() (sympy.polys.polytools.Poly method), 826
 cancel() (sympy.polys.rings.PolyElement method), 974
 canon_bp() (in module sympy.tensor.tensor), 1389
 canon_bp() (sympy.tensor.tensor.TensAdd method), 1386
 canon_bp() (sympy.tensor.tensor.TensMul method), 1387
 canonical_variables (sympy.core.basic.Basic attribute), 99
 canonicalize() (in module sympy.combinatorics.tensor_can), 289
 capture() (in module sympy.utilities.iterables), 1417
 cardinality (sympy.combinatorics.permutations.Permutation attribute), 213
 cardinality (sympy.combinatorics.subsets.Subset attribute), 268
 cartan_matrix() (sympy.liealgebras.root_system.RootSystem method), 704
 cartan_matrix() (sympy.liealgebras.type_a.TypeA method), 705
 cartan_matrix() (sympy.liealgebras.type_b.TypeB method), 707
 cartan_matrix() (sympy.liealgebras.type_c.TypeC method), 708
 cartan_matrix() (sympy.liealgebras.type_d.TypeD method), 709
 cartan_matrix() (sympy.liealgebras.type_e.TypeE method), 711
 cartan_matrix() (sympy.liealgebras.type_f.TypeF method), 712
 cartan_matrix() (sympy.liealgebras.type_g.TypeG method), 713
 CartanMatrix() (in module sympy.liealgebras.cartan_matrix), 717
 CartanType_generator (class in sympy.liealgebras.cartan_type),

- 716
casoratian() (in module sympy.matrices.dense), 758
- cast_check() (sympy.codegen.ast.Type method), 405
- cast_nocheck (sympy.codegen.ast.FloatBaseType attribute), 399
- cast_nocheck() (sympy.codegen.ast.FloatType method), 399
- Catalan (class in sympy.core.numbers), 151
- catalan (class in sympy.functions.combinatorial.numbers), 462
- Category (class in sympy.categories), 1740
- Cauchy() (in module sympy.stats), 1196
- Cbrt (class in sympy.codegen.cfunctions), 409
- cbrt() (in module sympy.functions.elementary.miscellaneous), 457
- ccode() (in module sympy.printing.ccode), 1024
- CCodeGen (class in sympy.utilities.codegen), 1400
- ceiling (class in sympy.functions.elementary.integers), 447
- center (sympy.geometry.ellipse.Ellipse attribute), 614
- center (sympy.geometry.polygon.RegularPolygon attribute), 636
- center() (sympy.combinatorics.perm_groups.PermGroup method), 237
- central_inertia (sympy.physics.mechanics.rigidbody.RigidBody attribute), 1624
- centralizer() (sympy.combinatorics.perm_groups.PermutationGroup method), 237
- centroid (sympy.geometry.polygon.Polygon attribute), 630
- centroid (sympy.geometry.polygon.RegularPolygon attribute), 636
- centroid() (in module sympy.geometry.util), 566
- CG (class in sympy.physics.quantum.cg), 1648
- cg_simp() (in module sympy.physics.quantum.cg), 1650
- CGate (class in sympy.physics.quantum.gate), 1690
- CGateS (class in sympy.physics.quantum.gate), 1693
- characteristic() (sympy.polys.domains.domain.Domain method), 905
- characteristic() (sympy.polys.domains.FiniteField method), 911
- chebyshev (class in sympy.functions.special.polynomials), 543
- chebyshev_poly() (in module sympy.polys.orthopolys), 873
- chebyshev_root (class in sympy.functions.special.polynomials), 545
- chebyshevu (class in sympy.functions.special.polynomials), 544
- chebyshevu_poly() (in module sympy.polys.orthopolys), 874
- chebyshevu_root (class in sympy.functions.special.polynomials), 545
- check_and_join() (in module sympy.crypto.crypto), 336
- check_assumptions() (in module sympy.solvers.solvers), 1308
- check_output() (sympy.utilities.runtests.SymPyOutputChecker method), 1448
- checkinfsol() (in module sympy.solvers.ode), 1237
- checkodesol() (in module sympy.solvers.ode), 1235
- checkpdesol() (in module sympy.solvers.pde), 1291
- checksol() (in module sympy.solvers.solvers), 1309
- Chi (class in sympy.functions.special.error_functions), 510
- ChiNoncentral() (in module sympy.stats), 1197
- ChiSquared() (in module sympy.stats), 1198
- cholesky() (sympy.matrices.matrices.MatrixBase method), 743
- cholesky() (sympy.matrices.sparse.SparseMatrix method), 769
- cholesky_solve() (sympy.matrices.matrices.MatrixBase method), 743
- Ci (class in sympy.functions.special.error_functions), 508
- Circle (class in sympy.geometry.ellipse), 624
- circumcenter (sympy.geometry.polygon.RegularPolygon attribute), 637
- circumcenter (sympy.geometry.polygon.Triangle attribute), 643
- circumcircle (sympy.geometry.polygon.RegularPolygon attribute), 637
- circumcircle (sympy.geometry.polygon.Triangle attribute), 643
- circumference (sympy.geometry.ellipse.Circle attribute), 625

- circumference (sympy.geometry.ellipse.Ellipse attribute), 614
- circumradius (sympy.geometry.polygon.RegularPolygon attribute), 637
- circumradius (sympy.geometry.polygon.Triangle attribute), 644
- CL (sympy.matrices.sparse.SparseMatrix attribute), 768
- class_key() (sympy.core.add.Add class method), 159
- class_key() (sympy.core.basic.Basic class method), 99
- classify_diop() (in module sympy.solvers.diophantine), 1320
- classify_ode() (in module sympy.solvers.ode), 1233
- classify_pde() (in module sympy.solvers.pde), 1290
- classof() (in module sympy.matrices.matrices), 754
- clear() (sympy.tensor.tensor._TensorManager method), 1375
- clear_denoms() (sympy.polys.polyclasses.DMP method), 919
- clear_denoms() (sympy.polys.polytools.Poly method), 826
- clebsch_gordan() (in module sympy.physics.wigner), 1497
- closing_angle() (sympy.geometry.line.Ray2D method), 601
- closure (sympy.sets.sets.Set attribute), 1132
- CMod (class in sympy.physics.quantum.shor), 1701
- cmplx (class in sympy.codegen.fnodes), 418
- CNOT (in module sympy.physics.quantum.gate), 1692
- CNotGate (class in sympy.physics.quantum.gate), 1692
- CodeBlock (class in sympy.codegen.ast), 396
- CodeGen (class in sympy.utilities.codegen), 1399
- codegen() (in module sympy.utilities.codegen), 1405
- CodePrinter (class in sympy.printing.codeprinter), 1054
- CodeWrapper (class in sympy.utilities.autowrap), 1392
- codomain (sympy.categories.CompositeMorphism attribute), 1738
- codomain (sympy.categories.Morphism attribute), 1736
- coeff() (sympy.core.expr.Expr method), 119
- coeff() (sympy.polys.rings.PolyElement method), 974
- coeff() (sympy.series.sequences.SeqBase method), 1115
- coeff_from() (sympy.polys.polytools.Poly method), 827
- coeff_mul() (sympy.series.sequences.EmptySequence method), 1118
- coeff_mul() (sympy.series.sequences.SeqBase method), 1115
- coeff_mul() (sympy.series.sequences.SeqFormula method), 1117
- coeff_mul() (sympy.series.sequences.SeqPer method), 1118
- coefficients (sympy.geometry.line.Line2D attribute), 600
- coeffs() (sympy.polys.numberfields.AlgebraicNumber method), 868
- coeffs() (sympy.polys.polyclasses.DMP method), 919
- coeffs() (sympy.polys.polytools.Poly method), 827
- coeffs() (sympy.polys.rings.PolyElement method), 975
- CoercionFailed (class in sympy.polys.polyerrors), 999
- cofactors() (in module sympy.polys.polytools), 812
- cofactors() (sympy.core.numbers.Number method), 138
- cofactors() (sympy.polys.domains.domain.Domain method), 905
- cofactors() (sympy.polys.polyclasses.DMP method), 919
- cofactors() (sympy.polys.polytools.Poly method), 828
- coherent_state() (in module sympy.physics.qho_1d), 1475
- Coin() (in module sympy.stats), 1191
- col_del() (sympy.matrices.dense.MutableDenseMatrix method), 762
- col_del() (sympy.matrices.sparse.MutableSparseMatrix method), 772
- col_join() (sympy.matrices.sparse.MutableSparseMatrix method), 772
- col_list() (sympy.matrices.sparse.SparseMatrix method), 769
- col_op() (sympy.matrices.dense.MutableDenseMatrix method), 763
- col_op() (sympy.matrices.sparse.MutableSparseMatrix method), 773
- col_swap() (sympy.matrices.dense.MutableDenseMatrix method), 763
- col_swap() (sympy.matrices.sparse.MutableSparseMatrix method), 773

- `collect()` (in module `sympy.simplify.radsimp`), 1163
- `collect()` (`sympy.core.expr.Expr` method), 121
- `collect_const()` (in module `sympy.simplify.radsimp`), 1166
- `collect_sqrt()` (in module `sympy.simplify.radsimp`), 1166
- `comb_explicit_rhs` (`sympy.physics.mechanics.system.SymbolicSystem` (class in `sympy.core.numbers`), 148 attribute), 1640
- `comb_implicit_mat` (`sympy.physics.mechanics.system.SymbolicSystem` (class in `sympy.sets.fancysets`), 1150 attribute), 1640
- `comb_implicit_rhs` (`sympy.physics.mechanics.system.SymbolicSystem` (class in `sympy.polys.rootoftools`), 870 attribute), 1641
- `combsimp()` (in module `sympy.simplify.combsimp`), 1173
- `combsimp()` (`sympy.core.expr.Expr` method), 121
- `comm_i2symbol()` (`sympy.tensor.tensor._TensorManager` method), 1375
- `comm_symbols2i()` (`sympy.tensor.tensor._TensorManager` method), 1375
- `CommaOperator` (class in `sympy.codegen.cnodes`), 413
- `Comment` (class in `sympy.codegen.ast`), 398
- `common_prefix()` (in module `sympy.utilities.iterables`), 1418
- `common_suffix()` (in module `sympy.utilities.iterables`), 1418
- `commutative`, 95
- `commutative` (`sympy.assumptions.ask.AssumptionKeys` method), 919 attribute), 1076
- `commutative_diagrams` (`sympy.categories.Category` attribute), 1741
- `Commutator` (class in `sympy.diffgeom`), 1759
- `Commutator` (class in `sympy.physics.quantum.commutator`), 1650
- `Commutator` (class in `sympy.physics.secondquant`), 1488
- `commutator()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 238
- `commutator()` (`sympy.combinatorics.permutations.Permutation` method), 213
- `commutes_with()` (`sympy.combinatorics.permutations.Permutation` method), 214
- `commutes_with()` (`sympy.tensor.tensor.TensorHead` method), 1383
- `compare()` (in module `sympy.series.gruntz`), 1108
- `compare()` (`sympy.core.basic.Basic` method), 99
- `Complement` (class in `sympy.sets.sets`), 1144
- `complement()` (`sympy.sets.sets.Set` method), 1133
- `complex`, 95
- `complex` (`sympy.assumptions.ask.AssumptionKeys` attribute), 1076
- `complex_elements` (`sympy.assumptions.ask.AssumptionKeys` attribute), 1077
- `ComplexInfinity` (class in `sympy.core.numbers`), 148
- `ComplexRegion` (class in `sympy.sets.fancysets`), 1150
- `ComplexRootOf` (class in `sympy.polys.rootoftools`), 870
- `ComplexSpace` (class in `sympy.physics.quantum.hilbert`), 1657
- `ComplexType` (class in `sympy.codegen.ast`), 398
- `components` (`sympy.categories.CompositeMorphism` attribute), 1739
- `components` (`sympy.vector.dyadic.Dyadic` attribute), 1796
- `components` (`sympy.vector.vector.Vector` attribute), 1793
- `compose()` (in module `sympy.polys.polytools`), 814
- `compose()` (`sympy.categories.Morphism` method), 1736
- `compose()` (`sympy.polys.polyclasses.DMP` method), 828
- `compose()` (`sympy.polys.polytools.Poly` method), 828
- `compose()` (`sympy.polys.rings.PolyRing` method), 973
- `composite`, 95
- `composite` (`sympy.assumptions.ask.AssumptionKeys` attribute), 1077
- `composite()` (in module `sympy.ntheory.generate`), 305
- `CompositeDomain` (class in `sympy.polys.domains.compositedomain`), 911
- `CompositeMorphism` (class in `sympy.categories`), 1738
- `Composition` (in module `sympy.ntheory.generate`), 306
- `compose_position()` (`sympy.holonomic.holonomic.HolonomicFunction` method), 660
- `ComputationFailed` (class in `sympy.polys.polyerrors`), 999
- `compute_explicit_form()` (`sympy.physics.mechanics.system.SymbolicSystem` method), 1641

- compute_fps() (in module sympy.series.formal), 1125
- compute_known_facts() (in module sympy.assumptions.ask), 1091
- compute_leading_term() (sympy.core.expr.Expr method), 121
- conclusions (sympy.categories.Diagram attribute), 1742
- condition_number() (sympy.matrices.matrices.MatrixBase method), 744
- ConditionalDomain (class in sympy.stats.rv), 1228
- conjugate (class in sympy.functions.elementary.complexes), 430
- conjugate (sympy.combinatorics.partitions.IntegerPartitions attribute), 205
- conjugate_gauss_beams() (in module sympy.physics.optics.gaussopt), 1713
- connect_to() (sympy.diffgeom.CoordSystem method), 1756
- conserve_mpmath_dps() (in module sympy.utilities.decorator), 1409
- const() (sympy.polys.rings.PolyElement method), 975
- constant_renumber() (in module sympy.solvers.ode), 1239
- constant_symbols() (sympy.physics.mechanics.system.SymbolicSystem method), 1641
- constantsimp() (in module sympy.solvers.ode), 1240
- construct_domain() (in module sympy.polys.constructor), 865
- contains() (sympy.combinatorics.perm_groups.PermutationGroup method), 238
- contains() (sympy.geometry.line.Line method), 590
- contains() (sympy.geometry.line.LinearEntity method), 582
- contains() (sympy.geometry.line.Ray method), 592
- contains() (sympy.geometry.line.Segment method), 595
- contains() (sympy.polys.agca.ideals.Ideal method), 894
- contains() (sympy.polys.agca.modules.Module method), 887
- contains() (sympy.polys.polytools.GroebnerBasis method), 862
- contains() (sympy.series.order.Order method), 1112
- contains() (sympy.sets.sets.Set method), 1133
- content() (in module sympy.polys.polytools), 813
- content() (sympy.polys.polyclasses.DMP method), 919
- content() (sympy.polys.polytools.Poly method), 828
- content() (sympy.polys.rings.PolyElement method), 975
- continued_fraction_convergents() (in module sympy.ntheory.continued_fraction), 330
- continued_fraction_iterator() (in module sympy.ntheory.continued_fraction), 331
- continued_fraction_periodic() (in module sympy.ntheory.continued_fraction), 332
- continued_fraction_reduce() (in module sympy.ntheory.continued_fraction), 332
- ContinueToken (class in sympy.codegen.ast), 398
- ContinuousDomain (class in sympy.stats.crv), 1229
- ContinuousPSpace (class in sympy.stats.crv), 1229
- ContinuousRV() (in module sympy.stats), 1220
- contract_metric() (sympy.tensor.tensor.TensAdd method), 1386
- contract_metric() (sympy.tensor.tensor.TensMul method), 1388
- contraction() (in module sympy.physics.secondquant), 1489
- controls (sympy.physics.quantum.gate.CGate attribute), 1690
- controls (sympy.physics.quantum.gate.CNotGate attribute), 1692
- convergence_statement (sympy.functions.special.hyper.hyper attribute), 531
- convert() (sympy.polys.agca.modules.FreeModule method), 888
- convert() (sympy.polys.agca.modules.Module method), 887
- convert() (sympy.polys.agca.modules.QuotientModule method), 896
- convert() (sympy.polys.agca.modules.SubModule method), 889
- convert() (sympy.polys.domains.domain.Domain method), 905
- convert() (sympy.polys.polyclasses.DMP method), 919

convert_from() (sympy.polys.domains.domain.Domain method), 905
 convert_to() (in module sympy.physics.units.util), 1513
 convert_to() (sympy.physics.units.quantities.Quantity method), 1513
 convert_to_native_paths() (in module sympy.utilities.runtests), 1448
 convert_xor() (in module sympy.parsing.sympy_parser), 1458
 convex_hull() (in module sympy.geometry.util), 565
 coord_function() (sympy.diffgeom.CoordSystem method), 1756
 coord_functions() (sympy.diffgeom.CoordSystem method), 1756
 coord_tuple_transform_to() (sympy.diffgeom.CoordSystem method), 1756
 coordinates (sympy.physics.mechanics.system.SymbolicSystem attribute), 1641
 CoordinateSym (class in sympy.physics.vector.frame), 1551
 coords() (sympy.diffgeom.Point method), 1757
 CoordSys3D (class in sympy.vector.coordsysrect), 1786
 CoordSystem (class in sympy.diffgeom), 1754
 copy() (sympy.matrices.matrices.MatrixBase method), 744
 copy() (sympy.polys.rings.PolyElement method), 975
 copy() (sympy.series.gruntz.SubSet method), 1110
 copyin_list() (sympy.matrices.dense.MutableDenseMatrix method), 763
 copyin_matrix() (sympy.matrices.dense.MutableDenseMatrix method), 764
 core() (in module sympy.ntheory.factor_), 319
 cornacchia() (in module sympy.solvers.diophantine), 1324
 corners (sympy.combinatorics.polyhedron.Polyhedron attribute), 261
 cos (class in sympy.functions.elementary.trigonometric), 433
 coset_factor() (sympy.combinatorics.perm_group.PermutationGroup method), 239
 coset_rank() (sympy.combinatorics.perm_groups.PermutationGroup method), 240
 coset_table() (sympy.combinatorics.perm_groups.PermutationGroup method), 240
 coset_transversal() (sympy.combinatorics.perm_groups.PermutationGroup method), 240
 cosh (class in sympy.functions.elementary.hyperbolic), 444
 cosine_transform() (in module sympy.integrals.transforms), 671
 cot (class in sympy.functions.elementary.trigonometric), 435
 coth (class in sympy.functions.elementary.hyperbolic), 444
 could_extract_minus_sign() (sympy.core.expr.Expr method), 121
 count() (sympy.core.basic.Basic method), 100
 count_complex_roots() (sympy.polys.polyclasses.DMP method), 919
 count_ops() (in module sympy.core.function), 192
 count_ops() (sympy.core.basic.Basic method), 100
 count_ops() (sympy.core.expr.Expr method), 121
 count_partitions() (sympy.utilities.enumerative.MultisetPart method), 1412
 count_real_roots() (sympy.polys.polyclasses.DMP method), 919
 count_roots() (in module sympy.polys.polytools), 818
 count_roots() (sympy.polys.polytools.Poly method), 828
 couple() (in module sympy.physics.quantum.spin), 1678
 CovarDerivativeOp (class in sympy.diffgeom), 763
 Covariance (class in sympy.stats), 1226
 CovarianceMatrix (in module sympy.stats), 1226
 coxeter_diagram() (sympy.liealgebras.weyl_group.WeylGroup method), 715
 create_expand_pow_optimization() (in module sympy.codegen.rewriting), 390
 create_new() (sympy.vector.coordsysrect.CoordSys3D method), 1786
 CreateBoson (class in sympy.physics.secondquant), 1482
 CreateCircuitGroup (in module sympy.physics.quantum.circuitplot), 1480
 CreateFermion (class in sympy.physics.secondquant), 1483
 cross() (in module sympy.physics.vector.functions), 1576

- `cross()` (sympy.matrices.matrices.MatrixBase method), 744
`cross()` (sympy.physics.vector.dyadic.Dyadic method), 1561
`cross()` (sympy.physics.vector.vector.Vector method), 1557
`cross()` (sympy.vector.deloperator.Del method), 1797
`cross()` (sympy.vector.dyadic.Dyadic method), 1796
`cross()` (sympy.vector.vector.Vector method), 1793
`crt()` (in module sympy.ntheory.modular), 322
`crt1()` (in module sympy.ntheory.modular), 323
`crt2()` (in module sympy.ntheory.modular), 323
`csc` (class in sympy.functions.elementary.trigonometric), 436
`csch` (class in sympy.functions.elementary.hyperbolic), 444
`cse()` (in module sympy.simplify.cse_main), 1174
`cse()` (sympy.codegen.ast.CodeBlock method), 397
`curl()` (in module sympy.physics.vector.fieldfunctions), 1579
`curl()` (in module sympy.vector), 1803
`current` (sympy.combinatorics.graycode.GrayCode attribute), 275
`Curve` (class in sympy.geometry.curve), 608
`CurvedMirror` (class in sympy.physics.optics.gaussopt), 1706
`CurvedRefraction` (class in sympy.physics.optics.gaussopt), 1705
`CXX11CodePrinter` (class in sympy.printing.cxxcode), 1027
`CXX98CodePrinter` (class in sympy.printing.cxxcode), 1027
`cxxcode()` (in module sympy.printing.cxxcode), 1027
`Cycle` (class in sympy.combinatorics.permutations), 230
`cycle_length()` (in module sympy.ntheory.generate), 305
`cycle_list()` (in module sympy.crypto.crypto), 337
`cycle_structure` (sympy.combinatorics.permutations.Permutation attribute), 214
`cycles` (sympy.combinatorics.permutations.Permutation attribute), 215
`cyclic()` (sympy.combinatorics.generators method), 231
`cyclic_form` (sympy.combinatorics.permutations.Permutation attribute), 215
`cyclic_form` (sympy.combinatorics.polyhedron.Polyhedron attribute), 262
`CyclicGroup()` (in module sympy.combinatorics.named_groups), 279
`cyclotomic_poly()` (in module sympy.polys.specialpolys), 873
`CythonCodeWrapper` (class in sympy.utilities.autowrap), 1392
- ## D
- `D` (sympy.matrices.matrices.MatrixBase attribute), 738
`D` (sympy.physics.optics.gaussopt.RayTransferMatrix attribute), 1704
`d()` (sympy.physics.quantum.spin.Rotation class method), 1671
`d()` (sympy.physics.quantum.spin.Rotation class method), 1671
`Dagger` (class in sympy.physics.quantum.dagger), 1652
`Dagger` (class in sympy.physics.secondquant), 1476
`Dagum()` (in module sympy.stats), 1199
`DataType` (class in sympy.utilities.codegen), 1399
`dcm()` (sympy.physics.vector.frame.ReferenceFrame method), 1553
`debug()` (in module sympy.utilities.misc), 1441
`debug_decorator()` (in module sympy.utilities.misc), 1441
`decimal_dig` (sympy.codegen.ast.FloatType attribute), 399
`decipher_affine()` (in module sympy.crypto.crypto), 339
`decipher_bifid()` (in module sympy.crypto.crypto), 344
`decipher_bifid5()` (in module sympy.crypto.crypto), 347
`decipher_bifid6()` (in module sympy.crypto.crypto), 348
`decipher_elgamal()` (in module sympy.crypto.crypto), 356, 358
`decipher_gm()` (in module sympy.crypto.crypto), 360
`decipher_hill()` (in module sympy.crypto.crypto), 343
`decipher_kid_rsa()` (in module sympy.crypto.crypto), 351

- decipher_rsa() (in module sympy.crypto.crypto), 350
- decipher_shift() (in module sympy.crypto.crypto), 338
- decipher_vigenere() (in module sympy.crypto.crypto), 342
- Declaration (class in sympy.codegen.ast), 398
- decode_morse() (in module sympy.crypto.crypto), 352
- decompose() (in module sympy.polys.polytools), 814
- decompose() (sympy.physics.quantum.gate.CGate method), 1690
- decompose() (sympy.physics.quantum.gate.SwapGate method), 1692
- decompose() (sympy.physics.quantum.qft.IQFT method), 1696
- decompose() (sympy.physics.quantum.qft.QFT method), 1696
- decompose() (sympy.polys.polyclasses.DMP method), 919
- decompose() (sympy.polys.polytools.Poly method), 829
- deduced() (sympy.codegen.ast.Variable class method), 408
- deflate() (sympy.polys.polyclasses.DMP method), 919
- deflate() (sympy.polys.polytools.Poly method), 829
- deflection() (sympy.physics.continuum_mechanics.beams.Beam method), 1726
- degree (sympy.combinatorics.perm_groups.PermutationsGroup attribute), 241
- degree() (in module sympy.polys.polytools), 803
- degree() (sympy.polys.polyclasses.DMP method), 919
- degree() (sympy.polys.polytools.Poly method), 829
- degree() (sympy.polys.rings.PolyElement method), 976
- degree_list() (in module sympy.polys.polytools), 803
- degree_list() (sympy.polys.polyclasses.DMP method), 919
- degree_list() (sympy.polys.polytools.Poly method), 829
- degrees() (sympy.polys.rings.PolyElement method), 976
- Del (class in sympy.vector.deloperator), 1797
- delete_doubles() (sympy.liealgebras.weyl_group.WeylGroup method), 715
- delta (sympy.functions.special.hyper.meijerg attribute), 534
- denom() (sympy.polys.domains.AlgebraicField method), 914
- denom() (sympy.polys.domains.domain.Domain method), 905
- denom() (sympy.polys.domains.ExpressionDomain method), 916
- denom() (sympy.polys.domains.FractionField method), 915
- denom() (sympy.polys.domains.ring.Ring method), 909
- denom() (sympy.polys.polyclasses.DMF method), 924
- density() (in module sympy.stats), 1223
- denote() (sympy.polys.agca.ideals.Ideal method), 894
- Derivative (class in sympy.core.function), 177
- derive_by_array() (in module sympy.tensor.array), 1363
- derived_series() (sympy.combinatorics.perm_groups.PermutationGroup method), 241
- derived_subgroup() (sympy.combinatorics.perm_groups.PermutationGroup method), 242
- descent() (in module sympy.solvers.diophantine), 1328
- descents() (sympy.combinatorics.permutations.Permutation method), 215
- deviation() (in module sympy.physics.optics.utils), 1716
- dh_private_key() (in module sympy.crypto.crypto), 356
- dh_public_key() (in module sympy.crypto.crypto), 356
- dh_shared_key() (in module sympy.crypto.crypto), 357
- diag() (in module sympy.matrices.dense), 755
- diagonal (sympy.assumptions.ask.AssumptionKeys attribute), 1077
- diagonal_solve() (sympy.matrices.matrices.MatrixBase method), 744
- Diagram (class in sympy.categories), 1741
- DiagramGrid (class in sympy.categories.diagram_drawing), 1745
- Dict (class in sympy.core.containers), 198
- dict_merge() (in module sympy.utilities.iterables), 1418
- Die() (in module sympy.stats), 1190
- DiePSpace (class in sympy.stats.frv_types), 1229
- diff() (in module sympy.core.function), 181
- diff() (sympy.holonomic.holonomic.HolonomicFunction method), 659
- diff() (sympy.physics.vector.vector.Vector method), 1558

- diff() (sympy.polys.polyclasses.DMP method), 919
- diff() (sympy.polys.polytools.Poly method), 830
- diff() (sympy.polys.rings.PolyElement method), 976
- difference_delta() (in module sympy.series.limitseq), 1130
- Differential (class in sympy.diffgeom), 1760
- DifferentialOperator (class in sympy.physics.quantum.operator), 1661
- differentiate_finite() (in module sympy.calculus.finite_diff), 1466
- dig (sympy.codegen.ast.FloatType attribute), 400
- digamma() (in module sympy.functions.special.gamma_functions), 487
- digit_2txt (in module sympy.printing.pretty.pretty_symbology), 1055
- digits() (in module sympy.ntheory.factor_), 320
- dihedral() (sympy.combinatorics.generators method), 232
- DihedralGroup() (in module sympy.combinatorics.named_groups), 280
- dim (sympy.physics.units.dimensions.DimensionSystem attribute), 1511
- dim (sympy.physics.units.unitsystem.UnitSystem attribute), 1512
- dim_can_vector() (sympy.physics.units.dimensions.DimensionSystem method), 1511
- dim_vector() (sympy.physics.units.dimensions.DimensionSystem method), 1511
- Dimension (class in sympy.physics.units.dimensions), 1509
- dimension (sympy.physics.quantum.hilbert.HilbertSpace attribute), 1657
- dimension() (in module sympy.codegen.fnodes), 418
- dimension() (sympy.liealgebras.type_a.TypeA method), 705
- dimension() (sympy.liealgebras.type_b.TypeB method), 707
- dimension() (sympy.liealgebras.type_c.TypeC method), 708
- dimension() (sympy.liealgebras.type_d.TypeD method), 710
- dimension() (sympy.liealgebras.type_e.TypeE method), 711
- dimension() (sympy.liealgebras.type_f.TypeF method), 712
- dimension() (sympy.liealgebras.type_g.TypeG method), 714
- DimensionSystem (class in sympy.physics.units.dimensions), 1510
- diop_bf_DN() (in module sympy.solvers.diophantine), 1325
- diop_DN() (in module sympy.solvers.diophantine), 1323
- diop_general_pythagorean() (in module sympy.solvers.diophantine), 1329
- diop_general_sum_of_even_powers() (in module sympy.solvers.diophantine), 1330
- diop_general_sum_of_squares() (in module sympy.solvers.diophantine), 1329
- diop_linear() (in module sympy.solvers.diophantine), 1321
- diop_quadratic() (in module sympy.solvers.diophantine), 1323
- diop_solve() (in module sympy.solvers.diophantine), 1321
- diop_ternary_quadratic() (in module sympy.solvers.diophantine), 1327
- diop_ternary_quadratic_normal() (in module sympy.solvers.diophantine), 1336
- diophantine() (in module sympy.solvers.diophantine), 1319
- Delta (class in sympy.functions.special.delta_functions), 475
- direction (sympy.geometry.line.LinearEntity attribute), 602
- direction_cosine (sympy.geometry.line.LinearEntity3D attribute), 603
- direction_cosine() (sympy.geometry.point.Point3D method), 577
- direction_ratio (sympy.geometry.line.LinearEntity3D attribute), 604
- direction_ratio() (sympy.geometry.point.Point3D method), 577
- DirectProduct() (in module sympy.combinatorics.group_constructs), 287
- dirichlet_eta (class in sympy.functions.special.zeta_functions), 526
- discrete_log() (in module sympy.ntheory.residue_ntheory), 330
- DiscreteUniform() (in module sympy.stats), 1190

`discriminant()` (in module `sympy.polys.polytools`), 808

`discriminant()` (`sympy.polys.polyclasses.DMP` method), 919

`discriminant()` (`sympy.polys.polytools.Poly` method), 830

`dispersion()` (in module `sympy.polys.dispersion`), 808, 881

`dispersion()` (`sympy.polys.polytools.Poly` method), 830

`dispersionset()` (in module `sympy.polys.dispersion`), 809, 880

`dispersionset()` (`sympy.polys.polytools.Poly` method), 831

`distance()` (`sympy.geometry.line.Line` method), 591

`distance()` (`sympy.geometry.line.Ray` method), 593

`distance()` (`sympy.geometry.line.Segment` method), 595

`distance()` (`sympy.geometry.plane.Plane` method), 651

`distance()` (`sympy.geometry.point.Point` method), 569

`distance()` (`sympy.geometry.polygon.Polygon` method), 630

`div()` (in module `sympy.polys.polytools`), 805

`div()` (`sympy.polys.domains.domain.Domain` method), 905

`div()` (`sympy.polys.domains.field.Field` method), 908

`div()` (`sympy.polys.domains.ring.Ring` method), 909

`div()` (`sympy.polys.polyclasses.DMP` method), 919

`div()` (`sympy.polys.polytools.Poly` method), 833

`div()` (`sympy.polys.rings.PolyElement` method), 976

`divergence` (`sympy.physics.optics.gaussopt.BeamParameter` attribute), 1709

`divergence()` (in module `sympy.physics.vector.fieldfunctions`), 1579

`divergence()` (in module `sympy.vector`), 1803

`divisible()` (in module `sympy.solvers.diophantine`), 1334

`divisor_count()` (in module `sympy.ntheory.factor_`), 315

`divisor_sigma` (class in `sympy.ntheory.factor_`), 318

`divisors()` (in module `sympy.ntheory.factor_`), 315

`DMF` (class in `sympy.polys.polyclasses`), 923

`DMP` (class in `sympy.polys.polyclasses`), 918

`dmp_abs()` (in module `sympy.polys.densearith`), 940

`dmp_add()` (in module `sympy.polys.densearith`), 940

`dmp_add_ground()` (in module `sympy.polys.densearith`), 938

`dmp_add_mul()` (in module `sympy.polys.densearith`), 941

`dmp_add_term()` (in module `sympy.polys.densearith`), 938

`dmp_apply_pairs()` (in module `sympy.polys.densebasic`), 937

`dmp_cancel()` (in module `sympy.polys.euclidtools`), 993

`dmp_clear_denoms()` (in module `sympy.polys.densetools`), 954

`dmp_compose()` (in module `sympy.polys.densetools`), 952

`dmp_content()` (in module `sympy.polys.euclidtools`), 992

`dmp_convert()` (in module `sympy.polys.densebasic`), 929

`dmp_copy()` (in module `sympy.polys.densebasic`), 928

`dmp_deflate()` (in module `sympy.polys.densebasic`), 934

`dmp_degree()` (in module `sympy.polys.densebasic`), 927

`dmp_degree_in()` (in module `sympy.polys.densebasic`), 927

`dmp_degree_list()` (in module `sympy.polys.densebasic`), 927

`dmp_diff()` (in module `sympy.polys.densetools`), 946

`dmp_diff_eval_in()` (in module `sympy.polys.densetools`), 948

`dmp_diff_in()` (in module `sympy.polys.densetools`), 946

`dmp_discriminant()` (in module `sympy.polys.euclidtools`), 989

`dmp_div()` (in module `sympy.polys.densearith`), 944

`dmp_eject()` (in module `sympy.polys.densebasic`), 936

`dmp_euclidean_prs()` (in module `sympy.polys.euclidtools`), 985

`dmp_eval()` (in module `sympy.polys.densetools`), 947

`dmp_eval_in()` (in module `sympy.polys.densetools`), 947

`dmp_eval_tail()` (in module `sympy.polys.densetools`), 947

dmp_exclude()	(in sympy.polys.densebasic), 935	module	dmp_inject()	(in sympy.polys.densebasic), 936	module
dmp_expand()	(in sympy.polys.densearith), 945	module	dmp_inner_gcd()	(in sympy.polys.euclidtools), 991	module
dmp_exquo()	(in sympy.polys.densearith), 945	module	dmp_inner_subresultants()	(in sympy.polys.euclidtools), 987	module
dmp_exquo_ground()	(in sympy.polys.densearith), 939	module	dmp_integrate()	(in sympy.polys.densetools), 946	module
dmp_ext_factor()	(in sympy.polys.factortools), 996	module	dmp_integrate_in()	(in sympy.polys.densetools), 946	module
dmp_factor_list()	(in sympy.polys.factortools), 997	module	dmp_invert()	(in sympy.polys.euclidtools), 984	module
dmp_factor_list_include()	(in sympy.polys.factortools), 997	module	dmp_irreducible_p()	(in sympy.polys.factortools), 997	module
dmp_ff_div()	(in sympy.polys.densearith), 943	module	dmp_l1_norm()	(in sympy.polys.densearith), 945	module
dmp_ff_prs_gcd()	(in sympy.polys.euclidtools), 990	module	dmp_LC()	(in sympy.polys.densebasic), 925	module
dmp_from_dict()	(in sympy.polys.densebasic), 933	module	dmp_lcm()	(in sympy.polys.euclidtools), 992	module
dmp_from_sympy()	(in sympy.polys.densebasic), 929	module	dmp_lift()	(in sympy.polys.densetools), 953	module
dmp_gcd()	(in sympy.polys.euclidtools), 991	module	dmp_list_terms()	(in sympy.polys.densebasic), 937	module
dmp_gcdex()	(in sympy.polys.euclidtools), 984	module	dmp_max_norm()	(in sympy.polys.densearith), 945	module
dmp_ground()	(in sympy.polys.densebasic), 931	module	dmp_mul()	(in sympy.polys.densearith), 941	module
dmp_ground_content()	(in sympy.polys.densetools), 950	module	dmp_mul_ground()	(in sympy.polys.densearith), 939	module
dmp_ground_extract()	(in sympy.polys.densetools), 951	module	dmp_mul_term()	(in sympy.polys.densearith), 938	module
dmp_ground_LC()	(in sympy.polys.densebasic), 926	module	dmp_multi_deflate()	(in sympy.polys.densebasic), 935	module
dmp_ground_monic()	(in sympy.polys.densetools), 949	module	dmp_neg()	(in sympy.polys.densearith), 940	module
dmp_ground_nth()	(in sympy.polys.densebasic), 930	module	dmp_negative_p()	(in sympy.polys.densebasic), 932	module
dmp_ground_p()	(in sympy.polys.densebasic), 931	module	dmp_nest()	(in sympy.polys.densebasic), 934	module
dmp_ground_primitive()	(in sympy.polys.densetools), 950	module	dmp_normal()	(in sympy.polys.densebasic), 929	module
dmp_ground_TC()	(in sympy.polys.densebasic), 926	module	dmp_nth()	(in sympy.polys.densebasic), 930	module
dmp_ground_trunc()	(in sympy.polys.densetools), 948	module	dmp_one()	(in sympy.polys.densebasic), 931	module
dmp_grounds()	(in sympy.polys.densebasic), 932	module	dmp_one_p()	(in sympy.polys.densebasic), 931	module
dmp_half_gcdex()	(in sympy.polys.euclidtools), 984	module	dmp_pdiv()	(in sympy.polys.densearith), 942	module
dmp_include()	(in sympy.polys.densebasic), 936	module	dmp_permute()	(in sympy.polys.densebasic), 934	module
dmp_inflate()	(in sympy.polys.densebasic), 935	module	dmp_pexquo()	(in sympy.polys.densearith), 943	module

dmp_positive_p()	(in module sympy.polys.densebasic), 932	dmp_terms_gcd()	(in module sympy.polys.densebasic), 936
dmp_pow()	(in module sympy.polys.densearith), 942	dmp_to_dict()	(in module sympy.polys.densebasic), 933
dmp_pquo()	(in module sympy.polys.densearith), 942	dmp_to_tuple()	(in module sympy.polys.densebasic), 929
dmp_prem()	(in module sympy.polys.densearith), 942	dmp_trial_division()	(in module sympy.polys.factortools), 993
dmp_primitive()	(in module sympy.polys.euclidtools), 992	dmp_true_LT()	(in module sympy.polys.densebasic), 926
dmp_primitive_prs()	(in module sympy.polys.euclidtools), 985	dmp_trunc()	(in module sympy.polys.densebasic), 948
dmp_prs_resultant()	(in module sympy.polys.euclidtools), 988	dmp_validate()	(in module sympy.polys.densebasic), 928
dmp_qq_collins_resultant()	(in module sympy.polys.euclidtools), 989	dmp_zero()	(in module sympy.polys.densebasic), 930
dmp_qq_heu_gcd()	(in module sympy.polys.euclidtools), 991	dmp_zero_p()	(in module sympy.polys.densebasic), 930
dmp_quo()	(in module sympy.polys.densearith), 944	dmp_zeros()	(in module sympy.polys.densebasic), 932
dmp_quo_ground()	(in module sympy.polys.densearith), 939	dmp_zz_collins_resultant()	(in module sympy.polys.euclidtools), 988
dmp_raise()	(in module sympy.polys.densebasic), 934	dmp_zz_diophantine()	(in module sympy.polys.factortools), 995
dmp_rem()	(in module sympy.polys.densearith), 944	dmp_zz_factor()	(in module sympy.polys.factortools), 996
dmp_resultant()	(in module sympy.polys.euclidtools), 989	dmp_zz_heu_gcd()	(in module sympy.polys.euclidtools), 990
dmp_revert()	(in module sympy.polys.densebasic), 954	dmp_zz_mignotte_bound()	(in module sympy.polys.factortools), 993
dmp_rr_div()	(in module sympy.polys.densearith), 943	dmp_zz_modular_resultant()	(in module sympy.polys.euclidtools), 988
dmp_rr_prs_gcd()	(in module sympy.polys.euclidtools), 989	dmp_zz_wang()	(in module sympy.polys.factortools), 995
dmp_slice()	(in module sympy.polys.densebasic), 937	dmp_zz_wang_hensel_lifting()	(in module sympy.polys.factortools), 995
dmp_sqr()	(in module sympy.polys.densearith), 942	dmp_zz_wang_lead_coeffs()	(in module sympy.polys.factortools), 995
dmp_strip()	(in module sympy.polys.densebasic), 928	dmp_zz_wang_non_divisors()	(in module sympy.polys.factortools), 995
dmp_sub()	(in module sympy.polys.densearith), 941	dmp_zz_wang_test_points()	(in module sympy.polys.factortools), 995
dmp_sub_ground()	(in module sympy.polys.densearith), 938	Do (class in sympy.codegen.fnodes), 414	
dmp_sub_mul()	(in module sympy.polys.densearith), 941	do_subs()	(sympy.series.gruntz.SubsSet method), 1110
dmp_sub_term()	(in module sympy.polys.densearith), 938	doctest()	(in module sympy.utilities.runtests), 1448
dmp_subresultants()	(in module sympy.polys.euclidtools), 987	doctest_depends_on()	(in module sympy.utilities.decorator), 1409
dmp_swap()	(in module sympy.polys.densebasic), 933	doit()	(sympy.core.basic.Basic method), 100
dmp_TC()	(in module sympy.polys.densebasic), 925	doit()	(sympy.integrals.Integral method), 692
		doit()	(sympy.physics.quantum.anticommutator.AntiCommutator method), 1648

`doit()` (sympy.physics.quantum.commutator.Commutator method), 1651
`doit()` (sympy.physics.secondquant.AntiSymmetrization method), 1494
`doit()` (sympy.physics.secondquant.Commutator method), 1488
`doit()` (sympy.physics.secondquant.NO method), 1491
`doit()` (sympy.physics.vector.dyadic.Dyadic method), 1561
`doit()` (sympy.physics.vector.vector.Vector method), 1558
`doit()` (sympy.series.limits.Limit method), 1106
`doit_numerically()` (sympy.core.function.Derivative method), 181
`Domain` (class in sympy.polys.domains.domain), 905
`domain` (sympy.categories.CompositeMorphism attribute), 1739
`domain` (sympy.categories.Morphism attribute), 1737
`domain` (sympy.polys.polytools.Poly attribute), 833
`domain_check()` (in module sympy.solvers.solveset), 1352
`DomainError` (class in sympy.polys.polyerrors), 999
`dominant()` (in module sympy.series.limitseq), 1130
`doprint()` (sympy.printing.mathematica.MCodePrinter class method), 1034
`doprint()` (sympy.printing.printer.Printer method), 1023
`doprint()` (sympy.printing.theanocode.TheanoPrinter method), 1044
`dot()` (in module sympy.physics.vector.functions), 1576
`dot()` (sympy.geometry.point.Point method), 569
`dot()` (sympy.matrices.matrices.MatrixBase method), 745
`dot()` (sympy.physics.vector.dyadic.Dyadic method), 1562
`dot()` (sympy.physics.vector.vector.Vector method), 1558
`dot()` (sympy.vector.deloperator.Del method), 1797
`dot()` (sympy.vector.dyadic.Dyadic method), 1796
`dot()` (sympy.vector.vector.Vector method), 1793
`dot_rot_grad_Ynm()` (in module sympy.physics.wigner), 1498
`dotprint()` (in module sympy.printing.dot), 1058
`dot_tensor_can_rep()` (in module sympy.combinatorics.tensor_can), 290
`draw()` (sympy.categories.diagram_drawing.XypicDiagram method), 1751
`drop()` (sympy.polys.rings.PolyRing method), 974
`drop_to_ground()` (sympy.polys.rings.PolyRing method), 974
`dsign` (class in sympy.codegen.fnodes), 419
`dsolve()` (in module sympy.solvers.ode), 1230
`dt()` (sympy.physics.vector.dyadic.Dyadic method), 1562
`dt()` (sympy.physics.vector.vector.Vector method), 1559
`dtype` (sympy.polys.agca.modules.FreeModule attribute), 888
`dtype` (sympy.polys.agca.modules.QuotientModule attribute), 896
`dtype` (sympy.polys.domains.AlgebraicField attribute), 914
`dtype` (sympy.polys.domains.ExpressionDomain attribute), 916
`dual` (sympy.physics.quantum.state.StateBase attribute), 1681
`dual()` (sympy.matrices.matrices.MatrixBase method), 745
`dual_class()` (sympy.physics.quantum.state.StateBase class method), 1681
`Dummy` (class in sympy.core.symbol), 135
`dummy_eq()` (sympy.core.basic.Basic method), 100
`DummyWrapper` (class in sympy.utilities.autowrap), 1392
`dump_c()` (sympy.utilities.autowrap.UfuncifyCodeWrapper method), 1392
`dump_c()` (sympy.utilities.codegen.CCodeGen method), 1401
`dump_code()` (sympy.utilities.codegen.CodeGen method), 1399
`dump_f95()` (sympy.utilities.codegen.FCodeGen method), 1402
`dump_h()` (sympy.utilities.codegen.CCodeGen method), 1401
`dump_h()` (sympy.utilities.codegen.FCodeGen method), 1402
`dump_jl()` (sympy.utilities.codegen.JuliaCodeGen method), 1403
`dump_m()` (sympy.utilities.codegen.OctaveCodeGen method), 1404
`dump_pyx()` (sympy.utilities.autowrap.CythonCodeWrapper method), 1392

`dump_rs()` (sympy.utilities.codegen.RustCodeGen method), 1404

`dup_content()` (in module sympy.polys.densetools), 949

`dup_cyclotomic_p()` (in module sympy.polys.factortools), 994

`dup_decompose()` (in module sympy.polys.densetools), 953

`dup_extract()` (in module sympy.polys.densetools), 951

`dup_gf_factor()` (in module sympy.polys.factortools), 997

`dup_lshift()` (in module sympy.polys.densearith), 939

`dup_mirror()` (in module sympy.polys.densetools), 951

`dup_monic()` (in module sympy.polys.densetools), 948

`dup_primitive()` (in module sympy.polys.densetools), 950

`dup_random()` (in module sympy.polys.densebasic), 937

`dup_real_imag()` (in module sympy.polys.densetools), 951

`dup_reverse()` (in module sympy.polys.densebasic), 928

`dup_rshift()` (in module sympy.polys.densearith), 940

`dup_scale()` (in module sympy.polys.densetools), 952

`dup_shift()` (in module sympy.polys.densetools), 952

`dup_sign_variations()` (in module sympy.polys.densetools), 953

`dup_transform()` (in module sympy.polys.densetools), 952

`dup_zz_cyclotomic_factor()` (in module sympy.polys.factortools), 994

`dup_zz_cyclotomic_poly()` (in module sympy.polys.factortools), 994

`dup_zz_factor()` (in module sympy.polys.factortools), 994

`dup_zz_factor_sqf()` (in module sympy.polys.factortools), 994

`dup_zz_hensel_lift()` (in module sympy.polys.factortools), 993

`dup_zz_hensel_step()` (in module sympy.polys.factortools), 993

`dup_zz_irreducible_p()` (in module sympy.polys.factortools), 994

`dup_zz_zassenhaus()` (in module sympy.polys.factortools), 994

`Dyadic` (class in sympy.physics.vector.dyadic), 1561

`dyadic` (class in sympy.physics.vector.dyadic), 1795

`dyn_implicit_mat` (sympy.physics.mechanics.system.Symbolic attribute), 1641

`dyn_implicit_rhs` (sympy.physics.mechanics.system.Symbolic attribute), 1641

`dynamic_symbols()` (sympy.physics.mechanics.system.Symbolic method), 1641

`dynamicsymbols()` (in module sympy.physics.vector), 1575

`dynkin_diagram()` (sympy.liealgebras.root_system.RootSystem method), 704

`DynkinDiagram` (in module sympy.liealgebras.dynkin_diagram), 716

E

`E()` (in module sympy.stats), 1221

`E1()` (in module sympy.functions.special.error_functions), 504

`E_n()` (in module sympy.physics.qho_1d), 1474

`E_nl()` (in module sympy.physics.hydrogen), 1469

`E_nl()` (in module sympy.physics.sho), 1475

`E_nl_dirac()` (in module sympy.physics.hydrogen), 1470

`EC()` (sympy.polys.polytools.Poly method), 822

`eccentricity` (sympy.geometry.ellipse.Ellipse attribute), 614

`edges` (sympy.combinatorics.polyhedron.Polyhedron attribute), 262

`edges()` (sympy.combinatorics.prufer.Prufer static method), 264

`egyptian_fraction()` (in module sympy.ntheory.egyptian_fraction), 334

`Ei` (class in sympy.functions.special.error_functions), 501

`Eijk()` (in module sympy.functions.special.tensor_functions), 553

`eject()` (sympy.polys.polyclasses.DMP method), 919

`eject()` (sympy.polys.polytools.Poly method), 833

`elastic_modulus` (sympy.physics.continuum_mechanics.beam.Beam attribute), 1727

`Element` (class in sympy.codegen.ast), 398

`element_order()` (sympy.liealgebras.weyl_group.WeylGroup method), 715

`elements` (sympy.combinatorics.perm_groups.PermutationGroup attribute), 242

`elgamal_private_key()` (in module sympy.crypto.crypto), 354

elgamal_public_key() (in module
 sympy.crypto.crypto), 354
 Ellipse (class in sympy.geometry.ellipse), 612
 elliptic_e (class in
 sympy.functions.special.elliptic_integrals),
 method), 1414
 536
 elliptic_f (class in
 sympy.functions.special.elliptic_integrals),
 method), 1414
 535
 elliptic_k (class in
 sympy.functions.special.elliptic_integrals),
 method), 1415
 534
 elliptic_pi (class in
 sympy.functions.special.elliptic_integrals),
 method), 1415
 536
 EM() (sympy.polys.polytools.Poly method),
 822
 emptyPrinter() (sympy.printing.repr.ReprPrinter
 method), 1050
 EmptySequence (class in
 sympy.series.sequences), 1118
 EmptySet (class in sympy.sets.sets), 1145
 encipher_affine() (in module
 sympy.crypto.crypto), 338
 encipher_bifid() (in module
 sympy.crypto.crypto), 344
 encipher_bifid5() (in module
 sympy.crypto.crypto), 346
 encipher_bifid6() (in module
 sympy.crypto.crypto), 348
 encipher_elgamal() (in module
 sympy.crypto.crypto), 355, 358
 encipher_gm() (in module
 sympy.crypto.crypto), 359
 encipher_hill() (in module
 sympy.crypto.crypto), 342
 encipher_kid_rsa() (in module
 sympy.crypto.crypto), 351
 encipher_rsa() (in module
 sympy.crypto.crypto), 350
 encipher_shift() (in module
 sympy.crypto.crypto), 337
 encipher_substitution() (in module
 sympy.crypto.crypto), 339
 encipher_vigenere() (in module
 sympy.crypto.crypto), 340
 encloses() (sympy.geometry.entity.GeometryEntity
 method), 561
 encloses_point() (sympy.geometry.ellipse.Ellipse
 method), 615
 encloses_point() (sympy.geometry.polygon.Polygon
 method), 630
 encloses_point() (sympy.geometry.polygon.RegularPolygon
 method), 637
 encode_morse() (in module
 sympy.crypto.crypto), 351
 end (sympy.sets.sets.Interval attribute), 1139
 enum_all() (sympy.utilities.enumerative.MultisetPartitionTra
 method), 1414
 enum_large() (sympy.utilities.enumerative.MultisetPartition
 method), 1414
 enum_range() (sympy.utilities.enumerative.MultisetPartition
 method), 1415
 enum_small() (sympy.utilities.enumerative.MultisetPartition
 method), 1415
 enumerate_states() (in module
 sympy.physics.quantum.represent),
 1669
 EPath (class in sympy.simplify.epathtools),
 1177
 epath() (in module
 sympy.simplify.epathtools), 1178
 eps (sympy.codegen.ast.FloatType attribute),
 400
 Eq (in module sympy.core.relational), 161
 Equality (class in sympy.core.relational), 162
 equals() (sympy.core.expr.Expr method), 121
 equals() (sympy.geometry.line.Line method),
 591
 equals() (sympy.geometry.line.Ray method),
 593
 equals() (sympy.geometry.line.Segment
 method), 596
 equals() (sympy.geometry.plane.Plane
 method), 652
 equals() (sympy.geometry.point.Point
 method), 569
 equals() (sympy.matrices.expressions.MatrixExpr
 method), 779
 equation() (sympy.geometry.ellipse.Circle
 method), 625
 equation() (sympy.geometry.ellipse.Ellipse
 method), 615
 equation() (sympy.geometry.line.Line2D
 method), 600
 equation() (sympy.geometry.line.Line3D
 method), 605
 equation() (sympy.geometry.plane.Plane
 method), 652
 Equivalent (class in sympy.logic.boolalg), 725
 equivalent() (in module
 sympy.solvers.diophantine), 1335
 erf (class in sympy.functions.special.error_functions),
 491
 erf2 (class in sympy.functions.special.error_functions),
 495
 erfai (class in sympy.functions.special.error_functions),
 497

erfc (class in sympy.functions.special.error_functions), 492
 erfcinv (class in sympy.functions.special.error_functions), 497
 erfi (class in sympy.functions.special.error_functions), 494
 erfinv (class in sympy.functions.special.error_functions), 496
 Erlang() (in module sympy.stats), 1200
 ET() (sympy.polys.polytools.Poly method), 822
 eta (sympy.functions.special.hyper.hyper attribute), 531
 euler (class in sympy.functions.combinatorial.numbers), 464
 euler_equations() (in module sympy.calculus.euler), 1460
 euler_maclaurin() (sympy.concrete.summations.Sum method), 363
 EulerGamma (class in sympy.core.numbers), 150
 eulerline (sympy.geometry.polygon.Triangle attribute), 644
 eval() (sympy.assumptions.assume.Predicate method), 1093
 eval() (sympy.functions.special.delta_functions.Delta class method), 477
 eval() (sympy.functions.special.delta_functions.Hypergeometric class method), 479
 eval() (sympy.functions.special.singularity_functions.SingularityFunction class method), 482
 eval() (sympy.functions.special.tensor_functions.KroneckerDelta class method), 554
 eval() (sympy.physics.secondquant.Commutator class method), 1489
 eval() (sympy.physics.secondquant.Dagger class method), 1477
 eval() (sympy.physics.secondquant.KroneckerDelta class method), 1478
 eval() (sympy.polys.polyclasses.DMP method), 919
 eval() (sympy.polys.polytools.Poly method), 833
 eval_controls() (sympy.physics.quantum.gate.Cgate method), 1690
 eval_expr() (in module sympy.parsing.sympy_parser), 1456
 eval_levicivita() (in module sympy.functions.special.tensor_functions), 553
 eval_zeta_function() (sympy.concrete.summations.Sum method), 364
 evalf() (sympy.geometry.point.Point method), 569
 evalf() (sympy.holonomic.holonomic.HolonomicFunction method), 662
 evalf() (sympy.polys.domains.domain.Domain method), 905
 evaluate_deltas() (in module sympy.physics.secondquant), 1493
 evaluate_pauli_product() (in module sympy.physics.paulialgebra), 1474
 EvaluationFailed (class in sympy.polys.polyerrors), 999
 even, 95
 evens() (sympy.assumptions.ask.AssumptionKeys attribute), 1078
 evolve() (sympy.geometry.ellipse.Ellipse method), 616
 ExactQuotientFailed (class in sympy.polys.polyerrors), 999
 exclude() (sympy.polys.polyclasses.DMP method), 919
 exclude() (sympy.polys.polytools.Poly method), 834
 exp (class in sympy.functions.elementary.exponential), 449
 exp() (sympy.matrices.matrices.MatrixBase method), 745
 Exp1 (class in sympy.core.numbers), 148
 exp2 (class in sympy.codegen.cfunctions), 148
 exp_re() (in module sympy.series.formal), 186
 expand() (in module sympy.core.function), 186
 expand() (sympy.core.expr.Expr method), 122
 expand_complex() (in module sympy.core.function), 194
 expand_func() (in module sympy.core.function), 193
 expand_log() (in module sympy.core.function), 193
 expand_mul() (in module sympy.core.function), 193
 expand_multinomial() (in module sympy.core.function), 194
 expand_power_base() (in module sympy.core.function), 195
 expand_power_exp() (in module sympy.core.function), 195
 expand_trig() (in module sympy.core.function), 194
 Expectation (class in sympy.stats), 1222
 expint (class in sympy.functions.special.error_functions), 502

- expm1 (class in `sympy.codegen.cfunctions`), 410
 Exponential() (in module `sympy.stats`), 1201
 Expr (class in `sympy.core.expr`), 111
 expr (`sympy.core.function.Lambda` attribute), 176
 expr (`sympy.core.function.Subs` attribute), 186
 expr (`sympy.physics.quantum.operator.DifferentialOperator` attribute), 1662
 expr (`sympy.physics.quantum.state.Wavefunction` attribute), 1686
 expr_free_symbols (`sympy.core.expr.Expr` attribute), 122
 expr_to_holonomic() (in module `sympy.holonomic.holonomic`), 664
 ExprCondPair (class in `sympy.functions.elementary.piecewise`), 450
 express() (in module `sympy.physics.vector.functions`), 1577
 express() (in module `sympy.vector`), 1802
 express() (`sympy.physics.vector.dyadic.Dyadic` method), 1562
 express() (`sympy.physics.vector.vector.Vector` method), 1559
 ExpressionDomain (class in `sympy.polys.domains`), 916
 ExpressionDomain.Expression (class in `sympy.polys.domains`), 916
 exquo() (in module `sympy.polys.polytools`), 806
 exquo() (`sympy.polys.domains.domain.Domain` method), 905
 exquo() (`sympy.polys.domains.field.Field` method), 908
 exquo() (`sympy.polys.domains.ring.Ring` method), 909
 exquo() (`sympy.polys.polyclasses.DMF` method), 924
 exquo() (`sympy.polys.polyclasses.DMP` method), 920
 exquo() (`sympy.polys.polytools.Poly` method), 834
 exquo_ground() (`sympy.polys.polyclasses.DMP` method), 920
 exquo_ground() (`sympy.polys.polytools.Poly` method), 834
 exradii (`sympy.geometry.polygon.Triangle` attribute), 644
 extend() (`sympy.ntheory.generate.Sieve` method), 299
 extend() (`sympy.physics.units.unitsystem.UnitSystem` method), 1512
 extend() (`sympy.plotting.plot.Plot` method), 1061
 extend_to_no() (`sympy.ntheory.generate.Sieve` method), 299
 extended_real (`sympy.assumptions.ask.AssumptionKeys` attribute), 1078
 F (class in `sympy.codegen.fnodes`), 415
 exterior_angle (`sympy.geometry.polygon.RegularPolygon` attribute), 638
 extract_additively() (`sympy.core.expr.Expr` method), 122
 extract_branch_factor() (`sympy.core.expr.Expr` method), 123
 extract_leading_order() (`sympy.core.add.Add` method), 160
 extract_multiplicatively() (`sympy.core.expr.Expr` method), 123
 extract_type_tens() (in module `sympy.physics.hep.gamma_matrices`), 1515
 ExtraneousFactors (class in `sympy.polys.polyerrors`), 999
 eye() (in module `sympy.matrices.dense`), 755
- ## F
- F (in module `sympy.physics.secondquant`), 1486
 F2PyCodeWrapper (class in `sympy.utilities.autowrap`), 1392
 faces (`sympy.combinatorics.polyhedron.Polyhedron` attribute), 262
 factor() (in module `sympy.polys.polytools`), 817
 factor() (`sympy.core.expr.Expr` method), 123
 factor_list() (in module `sympy.polys.polytools`), 816
 factor_list() (`sympy.polys.polyclasses.DMP` method), 920
 factor_list() (`sympy.polys.polytools.Poly` method), 835
 factor_list_include() (`sympy.polys.polyclasses.DMP` method), 920
 factor_list_include() (`sympy.polys.polytools.Poly` method), 835
 factor_terms() (in module `sympy.core.exprtools`), 201
 factorial (class in `sympy.functions.combinatorial.factorials`),

- 465
- factorial() (sympy.polys.domains.FractionField method), 915
- factorial() (sympy.polys.domains.PolynomialRing method), 912
- factorial2 (class in sympy.functions.combinatorial.factorials), 467
- factorial_notation() (in module sympy.parsing.sympy_parser), 1458
- factoring_visitor() (in module sympy.utilities.enumerative), 1411
- factorint() (in module sympy.ntheory.factor_), 312
- factors() (sympy.core.numbers.Rational method), 143
- FallingFactorial (class in sympy.functions.combinatorial.factorials), 468
- FBra (in module sympy.physics.secondquant), 1486
- fcode() (in module sympy.printing.fcode), 1030
- FCodeGen (class in sympy.utilities.codegen), 1401
- FCodePrinter (class in sympy.printing.fcode), 1032
- Fd (in module sympy.physics.secondquant), 1486
- fdiff() (sympy.codegen.cfunctions.Cbrt method), 410
- fdiff() (sympy.codegen.cfunctions.exp2 method), 410
- fdiff() (sympy.codegen.cfunctions.expm1 method), 411
- fdiff() (sympy.codegen.cfunctions.fma method), 411
- fdiff() (sympy.codegen.cfunctions.hypot method), 412
- fdiff() (sympy.codegen.cfunctions.log10 method), 412
- fdiff() (sympy.codegen.cfunctions.log1p method), 412
- fdiff() (sympy.codegen.cfunctions.log2 method), 413
- fdiff() (sympy.codegen.cfunctions.Sqrt method), 410
- fdiff() (sympy.core.function.Function method), 185
- fdiff() (sympy.functions.elementary.complexes.Abs method), 430
- fdiff() (sympy.functions.special.delta_functions.DiracDelta method), 478
- fdiff() (sympy.functions.special.delta_functions.Heaviside method), 480
- fdiff() (sympy.functions.special.singularity_functions.Singularity method), 482
- FDistribution() (in module sympy.stats), 1202
- fglm() (sympy.polys.polytools.GroebnerBasis method), 862
- fibonacci (class in sympy.functions.combinatorial.numbers), 468
- Field (class in sympy.polys.domains.field), 908
- field_isomorphism() (in module sympy.polys.numberfields), 867
- fill() (sympy.matrices.dense.MutableDenseMatrix method), 764
- fill() (sympy.matrices.sparse.MutableSparseMatrix method), 773
- filldedent() (in module sympy.utilities.misc), 1441
- filter_symbols() (in module sympy.utilities.iterables), 1418
- find() (sympy.core.basic.Basic method), 101
- find_DN() (in module sympy.solvers.diophantine), 1327
- find_dynamicsymbols() (in module sympy.physics.mechanics), 1643
- find_executable() (in module sympy.utilities.misc), 1442
- find_linear_recurrence() (sympy.series.sequences.SeqBase method), 1115
- finite, 95
- finite (sympy.assumptions.ask.AssumptionKeys attribute), 1078
- finite_diff_weights() (in module sympy.calculus.finite_diff), 1467
- FiniteDomain (class in sympy.stats.frv), 1229
- FiniteField (class in sympy.polys.domains), 911
- FinitePSpace (class in sympy.stats.frv), 1229
- FiniteRV() (in module sympy.stats), 1192
- FiniteSet (class in sympy.sets.sets), 1141
- FisherZ() (in module sympy.stats), 1203
- FixedBosonicBasis (class in sympy.physics.secondquant), 1487
- FKet (in module sympy.physics.secondquant), 1486
- FlagError (class in sympy.polys.polyerrors), 1000
- FlatMirror (class in sympy.physics.optics.gaussopt), 1705

FlatRefraction (class in sympy.physics.optics.gaussopt), 1704
 flatten() (in module sympy.utilities.iterables), 1418
 flatten() (sympy.categories.CompositeMorphism method), 1739
 flatten() (sympy.core.add.Add class method), 160
 flatten() (sympy.core.mul.Mul class method), 156
 Float (class in sympy.core.numbers), 138
 FloatBaseType (class in sympy.codegen.ast), 399
 FloatType (class in sympy.codegen.ast), 399
 floor (class in sympy.functions.elementary.integers), 448
 fma (class in sympy.codegen.cfunctions), 411
 foci (sympy.geometry.ellipse.Ellipse attribute), 616
 FockSpace (class in sympy.physics.quantum.hilbert), 1658
 FockState (class in sympy.physics.secondquant), 1485
 FockStateBosonBra (class in sympy.physics.secondquant), 1485
 FockStateBosonKet (class in sympy.physics.secondquant), 1485
 FockStateBra (class in sympy.physics.secondquant), 1485
 FockStateKet (class in sympy.physics.secondquant), 1485
 focus_distance (sympy.geometry.ellipse.Ellipse attribute), 616
 For (class in sympy.codegen.ast), 400
 forcing (sympy.physics.mechanics.kane.KanesMethod attribute), 1633
 forcing (sympy.physics.mechanics.lagrange.LagrangesMethod attribute), 1636
 forcing_full (sympy.physics.mechanics.kane.KanesMethod attribute), 1633
 forcing_full (sympy.physics.mechanics.lagrange.LagrangesMethod attribute), 1636
 form_lagranges_equations() (sympy.physics.mechanics.lagrange.LagrangesMethod method), 1636
 FormalPowerSeries (class in sympy.series.formal), 1123
 FortranReturn (class in sympy.codegen.fnodes), 415
 fourier_series() (in module sympy.series.fourier), 1122
 fourier_series() (sympy.core.expr.Expr method), 124
 fourier_transform() (in module sympy.integrals.transforms), 669
 FourierSeries (class in sympy.series.fourier), 1120
 fps() (in module sympy.series.formal), 1124
 fps() (sympy.core.expr.Expr method), 124
 frac (class in sympy.functions.elementary.integers), 448
 frac (in module sympy.printing.pretty.pretty_symbology), 1055
 frac_field() (sympy.polys.domains.domain.Domain method), 905
 frac_unify() (sympy.polys.polyclasses.DMF method), 924
 fraction() (in module sympy.simplify.radsimp), 1167
 FractionField (class in sympy.polys.domains), 915
 Frechet() (in module sympy.stats), 1203
 free_module() (sympy.polys.domains.ring.Ring method), 909
 free_symbols (sympy.core.basic.Basic attribute), 101
 free_symbols (sympy.geometry.curve.Curve attribute), 609
 free_symbols (sympy.integrals.Integral attribute), 692
 free_symbols (sympy.physics.quantum.operator.DifferentialOperator attribute), 1662
 free_symbols (sympy.physics.units.quantities.Quantity attribute), 1513
 free_symbols (sympy.polys.polytools.Poly attribute), 835
 free_symbols (sympy.polys.polytools.PurePoly attribute), 862
 free_symbols (sympy.series.sequences.SeqBase attribute), 1115
 free_symbols() (sympy.physics.vector.vector.Vector method), 1559
 free_symbols_in_domain (sympy.polys.polytools.Poly attribute), 836
 FreeModule (class in sympy.physics.optics.gaussopt), 1704
 FreeSpace (class in sympy.physics.optics.gaussopt), 1704
 frequency (sympy.physics.optics.waves.TWave attribute), 1721
 fresnelc (class in sympy.functions.special.error_functions), 500

FresnelIntegral (class in from_FractionField()
(sympy.functions.special.error_functions), (sympy.polys.domains.PolynomialRing
method), 913
498

fresnels (class in sympy.functions.special.error_functions), (sympy.polys.domains.PolynomialRing()
method), 913
498

from_AlgebraicField()
(sympy.polys.domains.domain.Domain method), 906

from_AlgebraicField()
(sympy.polys.domains.FractionField method), 915

from_AlgebraicField()
(sympy.polys.domains.IntegerRing method), 912

from_AlgebraicField()
(sympy.polys.domains.PolynomialRing method), 912

from_AlgebraicField()
(sympy.polys.domains.RationalField method), 914

from_ComplexField()
(sympy.polys.domains.domain.Domain method), 906

from_dict() (sympy.polys.polyclasses.DMP class method), 920

from_dict() (sympy.polys.polytools.Poly class method), 836

from_expr() (sympy.codegen.ast.Type class method), 406

from_expr() (sympy.polys.polytools.Poly class method), 836

from_ExpressionDomain()
(sympy.polys.domains.domain.Domain method), 906

from_ExpressionDomain()
(sympy.polys.domains.ExpressionDomain method), 917

from_FF_gmpy() (sympy.polys.domains.domain.Domain method), 906

from_FF_gmpy() (sympy.polys.domains.FiniteField method), 911

from_FF_python() (sympy.polys.domains.domain.Domain method), 906

from_FF_python() (sympy.polys.domains.FiniteField method), 911

from_FractionField()
(sympy.polys.domains.domain.Domain method), 906

from_FractionField()
(sympy.polys.domains.ExpressionDomain method), 917

from_FractionField()
(sympy.polys.domains.FractionField method), 915

from_hyper() (in module sympy.holonomic.holonomic), 664

from_index_summation()
(sympy.matrices.expressions.MatrixExpr static method), 780

from_inversion_vector()
(sympy.combinatorics.permutations.Permutation class method), 216

from_list() (sympy.polys.polyclasses.DMP class method), 920

from_list() (sympy.polys.polytools.Poly class method), 836

from_meijerg() (in module sympy.holonomic.holonomic), 664

from_poly() (sympy.polys.polytools.Poly class method), 836

from_PolynomialRing()
(sympy.polys.domains.domain.Domain method), 906

from_PolynomialRing()
(sympy.polys.domains.ExpressionDomain method), 917

from_PolynomialRing()
(sympy.polys.domains.FractionField method), 915

from_PolynomialRing()
(sympy.polys.domains.PolynomialRing method), 913

from_QQ_gmpy() (sympy.polys.domains.AlgebraicField method), 914

from_QQ_gmpy() (sympy.polys.domains.domain.Domain method), 906

from_QQ_gmpy() (sympy.polys.domains.ExpressionDomain method), 917

from_QQ_gmpy() (sympy.polys.domains.FiniteField method), 911

from_QQ_gmpy() (sympy.polys.domains.FractionField method), 915

from_QQ_gmpy() (sympy.polys.domains.PolynomialRing method), 913

from_QQ_python() (sympy.polys.domains.AlgebraicField method), 914

from_QQ_python() (sympy.polys.domains.domain.Domain method), 906

from_QQ_python() (sympy.polys.domains.ExpressionDomain method), 917

from_QQ_python() (sympy.polys.domains.FiniteField method), 911

from_QQ_python() (sympy.polys.domains.FractionField python() (sympy.polys.domains.ExpressionDomain
 method), 915 method), 917
 from_QQ_python() (sympy.polys.domains.PolynomialRing python() (sympy.polys.domains.FiniteField
 method), 913 method), 912
 from_real() (sympy.sets.fancysets.ComplexRegion from_ZZ_python() (sympy.polys.domains.FractionField
 class method), 1153 method), 915
 from_RealField() (sympy.polys.domains.AlgebraicField from_ZZ_python() (sympy.polys.domains.PolynomialRing
 method), 914 method), 913
 from_RealField() (sympy.polys.domains.domain.Domain from_ZZ_python() (sympy.core.basic.Basic class
 method), 906 method), 101
 from_RealField() (sympy.polys.domains.Expression from_ZZ_python() (sympy.combinatorics.permutations.Permutation
 method), 917 attribute), 216
 from_RealField() (sympy.polys.domains.FiniteField full_rank() (sympy.assumptions.ask.AssumptionKeys
 method), 911 attribute), 1079
 from_RealField() (sympy.polys.domains.FractionField full_eval() (sympy.tensor.tensor.TensAdd
 method), 915 method), 1386
 from_RealField() (sympy.polys.domains.PolynomialRing full_eval() (sympy.tensor.tensor.TensExpr
 method), 913 method), 1384
 from_rgs() (sympy.combinatorics.partitions.Partition function (sympy.core.basic.Basic attribute), 101
 class method), 203 func_field_modgcd() (in module
 from_sequence() (sympy.combinatorics.permutations.Permutation sympy.polys.modulargcd), 1003
 class method), 216 func_name() (in module sympy.utilities.misc),
 from_sympy() (sympy.polys.domains.AlgebraicField 1442
 method), 914 Function (class in sympy.core.function), 183
 from_sympy() (sympy.polys.domains.domain.Domain function (sympy.physics.quantum.operator.DifferentialOperator
 method), 906 attribute), 1663
 from_sympy() (sympy.polys.domains.Expression Domain exponentiation() (in module
 method), 917 sympy.parsing.sympy_parser), 1457
 from_sympy() (sympy.polys.domains.FiniteField FunctionCall (class in sympy.codegen.ast),
 method), 912 401
 from_sympy() (sympy.polys.domains.FractionField FunctionClass (class in sympy.core.function),
 method), 915 183
 from_sympy() (sympy.polys.domains.PolynomialRing FunctionDefinition (class in
 method), 913 sympy.codegen.ast), 401
 from_sympy() (sympy.polys.domains.RealField FunctionMatrix (class in
 method), 916 sympy.matrices.expressions), 783
 from_sympy_list() (sympy.polys.polyclasses.DMP FunctionPrototype (class in
 class method), 920 sympy.codegen.ast), 401
 from_ZZ_gmpy() (sympy.polys.domains.AlgebraicField functions (sympy.geometry.curve.Curve at-
 method), 914 tribute), 609
 from_ZZ_gmpy() (sympy.polys.domains.domain.Domain
 method), 906 **G**
 from_ZZ_gmpy() (sympy.polys.domains.Expression Domain (in module sympy.printing.pretty.pretty_symbolology),
 method), 917 1055
 from_ZZ_gmpy() (sympy.polys.domains.FiniteField g() (in module sympy.printing.pretty.pretty_symbolology),
 method), 911 1055
 from_ZZ_gmpy() (sympy.polys.domains.FractionField gamma (class in sympy.functions.special.gamma_functions),
 method), 915 483
 from_ZZ_gmpy() (sympy.polys.domains.PolynomialRing gamma() (in module sympy.stats), 1204
 method), 913 gamma trace() (in module
 from_ZZ_python() (sympy.polys.domains.AlgebraicField sympy.physics.hep.gamma_matrices),
 method), 914 1515
 from_ZZ_python() (sympy.polys.domains.domain.Domain gamma_inverse() (in module sympy.stats),
 method), 906 1205

`gammasimp()` (in module `sympy.core.expr.Expr` method), 124

`Gate` (class in module `sympy.physics.quantum.gate`), 1689

`gate` (`sympy.physics.quantum.gate.CGate` attribute), 1690

`gate` (`sympy.physics.quantum.gate.CNotGate` attribute), 1692

`gate_simp()` (in module `sympy.physics.quantum.gate`), 1693

`gate_sort()` (in module `sympy.physics.quantum.gate`), 1693

`gaunt()` (in module `sympy.physics.wigner`), 1498

`gauss_chebyshev_t()` (in module `sympy.integrals.quadrature`), 697

`gauss_chebyshev_u()` (in module `sympy.integrals.quadrature`), 698

`gauss_gen_laguerre()` (in module `sympy.integrals.quadrature`), 697

`gauss_hermite()` (in module `sympy.integrals.quadrature`), 696

`gauss_jacobi()` (in module `sympy.integrals.quadrature`), 699

`gauss_jordan_solve()` (`sympy.matrices.matrices.MatrixBase` method), 745

`gauss_laguerre()` (in module `sympy.integrals.quadrature`), 695

`gauss_legendre()` (in module `sympy.integrals.quadrature`), 694

`gaussian_conj()` (in module `sympy.physics.optics.gaussopt`), 1712

`gaussian_reduce()` (in module `sympy.solvers.diophantine`), 1337

`gcd()` (in module `sympy.polys.polytools`), 812

`gcd()` (`sympy.core.numbers.Number` method), 138

`gcd()` (`sympy.polys.domains.domain.Domain` method), 906

`gcd()` (`sympy.polys.domains.field.Field` method), 908

`gcd()` (`sympy.polys.domains.PolynomialRing` method), 913

`gcd()` (`sympy.polys.domains.RealField` method), 916

`gcd()` (`sympy.polys.polyclasses.DMP` method), 920

`gcd()` (`sympy.polys.polytools.Poly` method), 836

`gcd_list()` (in module `sympy.polys.polytools`), 812

`gcd_terms()` (in module `sympy.core.exprtools`), 200

`gcdex()` (in module `sympy.polys.polytools`), 807

`gcdex()` (`sympy.polys.domains.domain.Domain` method), 906

`gcdex()` (`sympy.polys.domains.PolynomialRing` method), 913

`gcdex()` (`sympy.polys.polyclasses.DMP` method), 920

`gcdex()` (`sympy.polys.polytools.Poly` method), 836

`Ge` (in module `sympy.core.relational`), 162

`gegenbauer` (class in module `sympy.functions.special.polynomials`), 542

`gegenbauer_poly()` (in module `sympy.polys.orthopolys`), 874

`gen` (`sympy.polys.polytools.Poly` attribute), 837

`gen` (`sympy.series.sequences.SeqBase` attribute), 1116

`generate()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 242

`generate_bell()` (in module `sympy.utilities.iterables`), 1419

`generate_derangements()` (in module `sympy.utilities.iterables`), 1420

`generate_dimino()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 243

`generate_gray()` (`sympy.combinatorics.graycode.GrayCode` method), 275

`generate_involutions()` (in module `sympy.utilities.iterables`), 1420

`generate_oriented_forest()` (in module `sympy.utilities.iterables`), 1421

`generate_schreier_sims()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 243

`generator_product()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 244

`generators` (`sympy.combinatorics.perm_groups.PermutationGroup` attribute), 244

`generators()` (`sympy.liealgebras.weyl_group.WeylGroup` method), 715

`GeneratorsError` (class in module `sympy.polys.polyerrors`), 1000

`GeneratorsNeeded` (class in module `sympy.polys.polyerrors`), 999

`Geometric()` (in module `sympy.stats`), 1192

`geometric_conj_ab()` (in module `sympy.physics.optics.gaussopt`), 1711

`geometric_conj_af()` (in module `sympy.physics.optics.gaussopt`), 1711
`geometric_conj_bf()` (in module `sympy.physics.optics.gaussopt`), 1712
`GeometricRay` (class in `sympy.physics.optics.gaussopt`), 1707
`GeometryEntity` (class in `sympy.geometry.entity`), 561
`get()` (`sympy.core.containers.Dict` method), 198
`get_adjacency_distance()` (`sympy.combinatorics.permutations.Permutation` method), 216
`get_adjacency_matrix()` (`sympy.combinatorics.permutations.Permutation` method), 217
`get_basis()` (in module `sympy.physics.quantum.represent`), 1669
`get_class()` (in module `sympy.utilities.source`), 1454
`get_comm()` (`sympy.tensor.tensor._TensorManager` method), 1375
`get_contraction_structure()` (in module `sympy.tensor.index_methods`), 1372
`get_default_datatype()` (in module `sympy.utilities.codegen`), 1399
`get_domain()` (`sympy.polys.polytools.Poly` method), 837
`get_exact()` (`sympy.polys.domains.domain.Domain` method), 906
`get_exact()` (`sympy.polys.domains.RealField` method), 916
`get_field()` (`sympy.polys.domains.domain.Domain` method), 906
`get_field()` (`sympy.polys.domains.ExpressionDomain` method), 917
`get_field()` (`sympy.polys.domains.field.Field` method), 909
`get_field()` (`sympy.polys.domains.FiniteField` method), 912
`get_field()` (`sympy.polys.domains.IntegerRing` method), 912
`get_field()` (`sympy.polys.domains.PolynomialRing` method), 913
`get_free_indices()` (`sympy.tensor.tensor.Tensors` method), 1388
`get_indices()` (in module `sympy.tensor.index_methods`), 1374
`get_indices()` (`sympy.tensor.tensor.Tensors` method), 1389
`get_interface()` (`sympy.utilities.codegen.FCodeGen` method), 1402
`get_matrix()` (`sympy.tensor.tensor.Tensors` method), 1384
`get_mod_func()` (in module `sympy.utilities.source`), 1454
`get_modulus()` (`sympy.polys.polytools.Poly` method), 837
`get_motion_params()` (in module `sympy.physics.vector.functions`), 1570
`get_period()` (`sympy.functions.special.hyper.meijerg` method), 534
`get_permuted()` (`sympy.physics.secondquant.PermutationOp` method), 1496
`get_positional_distance()` (`sympy.combinatorics.permutations.Permutation` method), 217
`get_precedence_distance()` (`sympy.combinatorics.permutations.Permutation` method), 218
`get_precedence_matrix()` (`sympy.combinatorics.permutations.Permutation` method), 218
`get_prototype()` (`sympy.utilities.codegen.CCodeGen` method), 1401
`get_prototype()` (`sympy.utilities.codegen.RustCodeGen` method), 1405
`get_resource()` (in module `sympy.utilities.pkgdata`), 1444
`get_ring()` (`sympy.polys.domains.AlgebraicField` method), 914
`get_ring()` (`sympy.polys.domains.domain.Domain` method), 906
`get_ring()` (`sympy.polys.domains.ExpressionDomain` method), 917
`get_ring()` (`sympy.polys.domains.field.Field` method), 909
`get_ring()` (`sympy.polys.domains.FractionField` method), 915
`get_ring()` (`sympy.polys.domains.RealField` method), 916
`get_ring()` (`sympy.polys.domains.ring.Ring` method), 910
`get_segments()` (`sympy.plotting.plot.LineOver1DRangeSeries` method), 1071
`get_segments()` (`sympy.plotting.plot.Parametric2DLineSeries` method), 1071
`get_subNO()` (`sympy.physics.secondquant.NO` method), 1491
`get_subset_from_bitstring()` (`sympy.combinatorics.graycode` method), 278

`get_symmetric_group_sgs()` (in module `sympy.combinatorics.tensor_can`), 293

`get_sympy_dir()` (in module `sympy.utilities.runtests`), 1449

`get_target_matrix()` (`sympy.physics.quantum.gate.Gate` method), 1689

`get_target_matrix()` (`sympy.physics.quantum.gate.UGate` method), 1690

`getn()` (`sympy.core.expr.Expr` method), 124

`getO()` (`sympy.core.expr.Expr` method), 124

`gf_add()` (in module `sympy.polys.galoistools`), 959

`gf_add_ground()` (in module `sympy.polys.galoistools`), 958

`gf_add_mul()` (in module `sympy.polys.galoistools`), 960

`gf_berlekamp()` (in module `sympy.polys.galoistools`), 969

`gf_cofactors()` (in module `sympy.polys.galoistools`), 964

`gf_compose()` (in module `sympy.polys.galoistools`), 965

`gf_compose_mod()` (in module `sympy.polys.galoistools`), 966

`gf_crt()` (in module `sympy.polys.galoistools`), 954

`gf_crt1()` (in module `sympy.polys.galoistools`), 955

`gf_crt2()` (in module `sympy.polys.galoistools`), 955

`gf_csolve()` (in module `sympy.polys.galoistools`), 971

`gf_degree()` (in module `sympy.polys.galoistools`), 956

`gf_diff()` (in module `sympy.polys.galoistools`), 965

`gf_div()` (in module `sympy.polys.galoistools`), 961

`gf_eval()` (in module `sympy.polys.galoistools`), 965

`gf_expand()` (in module `sympy.polys.galoistools`), 961

`gf_exquo()` (in module `sympy.polys.galoistools`), 962

`gf_factor()` (in module `sympy.polys.galoistools`), 970

`gf_factor_sqf()` (in module `sympy.polys.galoistools`), 969

`gf_from_dict()` (in module `sympy.polys.galoistools`), 957

`gf_from_int_poly()` (in module `sympy.polys.galoistools`), 958

`gf_gcd()` (in module `sympy.polys.galoistools`), 963

`gf_gcdex()` (in module `sympy.polys.galoistools`), 964

`gf_int()` (in module `sympy.polys.galoistools`), 955

`gf_irreducible()` (in module `sympy.polys.galoistools`), 966

`gf_irreducible_p()` (in module `sympy.polys.galoistools`), 967

`gf_LC()` (in module `sympy.polys.galoistools`), 956

`gf_lcm()` (in module `sympy.polys.galoistools`), 963

`gf_lshift()` (in module `sympy.polys.galoistools`), 962

`gf_monic()` (in module `sympy.polys.galoistools`), 964

`gf_mul()` (in module `sympy.polys.galoistools`), 960

`gf_mul_ground()` (in module `sympy.polys.galoistools`), 959

`gf_multi_eval()` (in module `sympy.polys.galoistools`), 965

`gf_neg()` (in module `sympy.polys.galoistools`), 958

`gf_normal()` (in module `sympy.polys.galoistools`), 957

`gf_pow()` (in module `sympy.polys.galoistools`), 962

`gf_pow_mod()` (in module `sympy.polys.galoistools`), 963

`gf_Qbasis()` (in module `sympy.polys.galoistools`), 968

`gf_Qmatrix()` (in module `sympy.polys.galoistools`), 968

`gf_quo()` (in module `sympy.polys.galoistools`), 961

`gf_quo_ground()` (in module `sympy.polys.galoistools`), 959

`gf_random()` (in module `sympy.polys.galoistools`), 966

`gf_rem()` (in module `sympy.polys.galoistools`), 961

`gf_rshift()` (in module `sympy.polys.galoistools`), 962

`gf_shoup()` (in module `sympy.polys.galoistools`), 969

`gf_sqf_list()` (in module `sympy.polys.galoistools`), 967

`gf_sqf_p()` (in module `sympy.polys.galoistools`), 967

`gf_sqf_part()` (in module `sympy.polys.galoistools`), 967

- gf_sqr() (in module sympy.polys.galoistools), 960
- gf_strip() (in module sympy.polys.galoistools), 956
- gf_sub() (in module sympy.polys.galoistools), 959
- gf_sub_ground() (in module sympy.polys.galoistools), 959
- gf_sub_mul() (in module sympy.polys.galoistools), 960
- gf_TC() (in module sympy.polys.galoistools), 956
- gf_to_dict() (in module sympy.polys.galoistools), 957
- gf_to_int_poly() (in module sympy.polys.galoistools), 958
- gf_trace_map() (in module sympy.polys.galoistools), 966
- gf_trunc() (in module sympy.polys.galoistools), 957
- gf_value() (in module sympy.polys.galoistools), 970
- gf_zassenhaus() (in module sympy.polys.galoistools), 969
- gff() (in module sympy.polys.polytools), 815
- gff_list() (in module sympy.polys.polytools), 815
- gff_list() (sympy.polys.polyclasses.DMP method), 920
- gff_list() (sympy.polys.polytools.Poly method), 837
- given() (in module sympy.stats), 1223
- gm_private_key() (in module sympy.crypto.crypto), 359
- gm_public_key() (in module sympy.crypto.crypto), 359
- GMPYFiniteField (class in sympy.polys.domains), 918
- GMPYIntegerRing (class in sympy.polys.domains), 918
- GMPYRationalField (class in sympy.polys.domains), 918
- GoldenRatio (class in sympy.core.numbers), 151
- gosper_normal() (in module sympy.concrete.gosper), 372
- gosper_sum() (in module sympy.concrete.gosper), 373
- gosper_term() (in module sympy.concrete.gosper), 373
- goto (class in sympy.codegen.cnodes), 414
- GoTo (class in sympy.codegen.fnodes), 415
- gouy (sympy.physics.optics.gaussopt.BeamParameter attribute), 1709
- GradedLexOrder (class in sympy.polys.orderings), 869
- gradient() (in module sympy.physics.vector.fieldfunctions), 1580
- gradient() (in module sympy.vector), 1804
- gradient() (sympy.vector.deloperator.Del method), 1798
- GramSchmidt() (in module sympy.matrices.dense), 757
- gray_to_bin() (sympy.combinatorics.graycode method), 277
- GrayCode (class in sympy.combinatorics.graycode), 274
- graycode_subsets() (sympy.combinatorics.graycode method), 278
- GreaterThan (class in sympy.core.relational), 163
- greek_letters (in module sympy.printing.pretty.pretty_symbology), 1055
- groebner() (in module sympy.polys.groebnertools), 997
- groebner() (in module sympy.polys.polytools), 820
- GroebnerBasis (class in sympy.polys.polytools), 862
- ground_roots() (in module sympy.polys.polytools), 819
- ground_roots() (sympy.polys.polytools.Poly method), 838
- group() (in module sympy.utilities.iterables), 1421
- group_name() (sympy.liealgebras.weyl_group.WeylGroup method), 716
- group_order() (sympy.liealgebras.weyl_group.WeylGroup method), 716
- grover_iteration() (in module sympy.physics.quantum.grover), 1695
- gruntz() (in module sympy.series.gruntz), 1108
- Gt (in module sympy.core.relational), 162
- ## H
- H (in module sympy.physics.quantum.gate), 1692
- HadamardGate (class in sympy.physics.quantum.gate), 1691
- Half (class in sympy.core.numbers), 146
- half_gcdex() (in module sympy.polys.polytools), 806

[half_gcdex\(\)](#) (sympy.polys.domains.domain.Domain method), [906](#)
[half_gcdex\(\)](#) (sympy.polys.polyclasses.DMP method), [920](#)
[half_gcdex\(\)](#) (sympy.polys.polytools.Poly method), [838](#)
[half_per\(\)](#) (sympy.polys.polyclasses.DMF method), [924](#)
[hankel1](#) (class in sympy.functions.special.bessel), [514](#)
[hankel2](#) (class in sympy.functions.special.bessel), [515](#)
[hankel_transform\(\)](#) (in module sympy.integrals.transforms), [672](#)
[harmonic](#) (class in sympy.functions.combinatorial.numbers), [469](#)
[has\(\)](#) (sympy.core.basic.Basic method), [101](#)
[has_dups\(\)](#) (in module sympy.utilities.iterables), [1421](#)
[has_integer_powers\(\)](#) (sympy.physics.units.dimensions.Dimension method), [1510](#)
[has_only_gens\(\)](#) (sympy.polys.polytools.Poly method), [838](#)
[has_q_annihilators](#) (sympy.physics.secondquant.NO attribute), [1491](#)
[has_q_creators](#) (sympy.physics.secondquant.NO attribute), [1492](#)
[has_variety\(\)](#) (in module sympy.utilities.iterables), [1422](#)
[Heaviside](#) (class in sympy.functions.special.delta_functions), [479](#)
[height](#) (sympy.categories.diagram_drawing.DiagramGrid attribute), [1747](#)
[height](#) (sympy.physics.optics.gaussopt.GeometricRay attribute), [1708](#)
[height\(\)](#) (sympy.polys.agca.ideals.Ideal method), [894](#)
[height\(\)](#) (sympy.printing.pretty.stringpict.StringPict method), [1056](#)
[hermite](#) (class in sympy.functions.special.polynomials), [547](#)
[hermite_poly\(\)](#) (in module sympy.polys.orthopolys), [874](#)
[hermitian](#), [96](#)
[hermitian](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1079](#)
[HermitianOperator](#) (class in sympy.physics.quantum.operator), [1660](#)
[hessian\(\)](#) (in module sympy.matrices.dense), [757](#)
[HeuristicGCDFailed](#) (class in sympy.polys.polyerrors), [999](#)
[highest_root\(\)](#) (sympy.liealgebras.type_a.TypeA method), [706](#)
[HilbertSpace](#) (class in sympy.physics.quantum.hilbert), [1657](#)
[hobj\(\)](#) (in module sympy.printing.pretty.pretty_symbology), [1055](#)
[holzer\(\)](#) (in module sympy.solvers.diophantine), [1337](#)
[hom\(\)](#) (sympy.categories.Diagram method), [1743](#)
[homogeneous_order\(\)](#) (in module sympy.solvers.ode), [1235](#)
[homogeneous_order\(\)](#) (sympy.polys.polyclasses.DMP method), [920](#)
[homogeneous_order\(\)](#) (sympy.polys.polytools.Poly method), [838](#)
[homogenize\(\)](#) (sympy.polys.polyclasses.DMP method), [920](#)
[homogenize\(\)](#) (sympy.polys.polytools.Poly method), [839](#)
[homomorphism\(\)](#) (in module sympy.polys.agca.homomorphisms), [899](#)
[HomomorphismFailed](#) (class in sympy.polys.polyerrors), [999](#)
[horner\(\)](#) (in module sympy.polys.polyfuncs), [864](#)
[hradius](#) (sympy.geometry.ellipse.Ellipse attribute), [617](#)
[hyper](#) (class in sympy.functions.special.hyper), [530](#)
[hyper_algorithm\(\)](#) (in module sympy.series.formal), [1129](#)
[hyper_re\(\)](#) (in module sympy.series.formal), [1128](#)
[HyperbolicFunction](#) (class in sympy.functions.elementary.hyperbolic), [443](#)
[hyperexpand\(\)](#) (in module sympy.simplify.hyperexpand), [1176](#)
[hyperfocal_distance\(\)](#) (in module sympy.physics.optics.utils), [1719](#)
[Hypergeometric\(\)](#) (in module sympy.stats), [1191](#)
[hypersimilar\(\)](#) (in module sympy.simplify.simplify), [1159](#)
[hypersimp\(\)](#) (in module sympy.simplify.simplify), [1159](#)

- hypot (class in sympy.codegen.cfunctions), 411
- |
- ibin() (in module sympy.utilities.iterables), 1422
- Ideal (class in sympy.polys.agca.ideals), 893
- ideal() (sympy.polys.domains.ring.Ring method), 910
- Identity (class in sympy.matrices.expressions), 783
- identity (sympy.combinatorics.perm_groups.PermGroup attribute), 244
- identity_hom() (sympy.polys.agca.modules.FreeModule method), 888
- identity_hom() (sympy.polys.agca.modules.Module method), 887
- identity_hom() (sympy.polys.agca.modules.QuotientModule method), 896
- identity_hom() (sympy.polys.agca.modules.SubModule method), 890
- IdentityFunction (class in sympy.functions.elementary.miscellaneous), 452
- IdentityGate (class in sympy.physics.quantum.gate), 1690
- IdentityMorphism (class in sympy.categories), 1740
- IdentityOperator (class in sympy.physics.quantum.operator), 1660
- Idx (class in sympy.tensor.indexed), 1367
- igcd() (in module sympy.core.numbers), 144
- ilcm() (in module sympy.core.numbers), 144
- im (class in sympy.functions.elementary.complexes), 428
- image() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 900
- ImageSet (class in sympy.sets.fancysets), 1148
- imageset() (in module sympy.sets.sets), 1137
- imaginary, 95
- imaginary (sympy.assumptions.ask.AssumptionKeys attribute), 1079
- ImaginaryUnit (class in sympy.core.numbers), 149
- ImmutableDenseMatrix (class in sympy.matrices.immutable), 777
- ImmutableDenseNDimArray (class in sympy.tensor.array), 1363
- ImmutableSparseMatrix (class in sympy.matrices.immutable), 776
- ImmutableSparseNDimArray (class in sympy.tensor.array), 1363
- implemented_function() (in module sympy.utilities.lambdify), 1437
- implicit_application() (in module sympy.parsing.sympy_parser), 1457
- implicit_multiplication() (in module sympy.parsing.sympy_parser), 1457
- implicit_multiplication_application() (in module sympy.parsing.sympy_parser), 1457
- ImplicitSeries (class in sympy.plotting.plot_implicit), 1072
- ImplicitGroup (class in sympy.codegen.fnodes), 416
- Module (class in sympy.logic.boolalg), 724
- imul_num() (sympy.polys.rings.PolyElement method), 976
- in_terms_of_generators() (sympy.polys.agca.modules.SubModule method), 890
- incenter (sympy.geometry.polygon.Triangle attribute), 645
- incircle (sympy.geometry.polygon.RegularPolygon attribute), 638
- incircle (sympy.geometry.polygon.Triangle attribute), 645
- inclusion_hom() (sympy.polys.agca.modules.SubModule method), 890
- indent_code() (sympy.printing.ccode.C89CodePrinter method), 1024
- indent_code() (sympy.printing.fcode.FCodePrinter method), 1032
- indent_code() (sympy.printing.jscode.JavascriptCodePrinter method), 1035
- indent_code() (sympy.printing.julia.JuliaCodePrinter method), 1037
- indent_code() (sympy.printing.octave.OctaveCodePrinter method), 1039
- indent_code() (sympy.printing.rcode.RCodePrinter method), 1028
- indent_code() (sympy.printing.rust.RustCodePrinter method), 1042
- index() (sympy.combinatorics.permutations.PermutationKeys method), 218
- index() (sympy.core.containers.Tuple method), 197
- index() (sympy.physics.secondquant.FixedBosonicBasis method), 1487
- index() (sympy.physics.secondquant.VarBosonicBasis method), 1487
- index() (sympy.polys.rings.PolyRing method), 974
- Indexed (class in sympy.tensor.indexed), 1369
- IndexedBase (class in sympy.tensor.indexed), 1370

indices (sympy.tensor.indexed.Indexed attribute), 1369
 indices_contain_equal_information (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 554
 indices_contain_equal_information (sympy.physics.secondquant.KroneckerDelta attribute), 1478
 inertia() (in module sympy.physics.mechanics.functions), 1626
 inertia_of_point_mass() (in module sympy.physics.mechanics.functions), 1627
 inf (sympy.sets.sets.Set attribute), 1133
 infinite, 95
 infinite (sympy.assumptions.ask.AssumptionKeys attribute), 1080
 infinite (sympy.series.formal.FormalPowerSeries attribute), 1123
 infinitesimal (sympy.assumptions.ask.AssumptionKeys attribute), 1080
 infinitesimals() (in module sympy.solvers.ode), 1236
 Infinity (class in sympy.core.numbers), 147
 infinity (sympy.assumptions.ask.AssumptionKeys attribute), 1080
 init_printing() (in module sympy.physics.vector.printing), 1572
 inject() (sympy.polys.domains.compositedomain.Compositedomain method), 911
 inject() (sympy.polys.domains.domain.Domain method), 906
 inject() (sympy.polys.domains.simplesdomain.Simplesdomain method), 911
 inject() (sympy.polys.polyclasses.DMP method), 920
 inject() (sympy.polys.polytools.Poly method), 839
 InnerProduct (class in sympy.physics.quantum.innerproduct), 1653
 InnerProduct (class in sympy.physics.secondquant), 1486
 inradius (sympy.geometry.polygon.RegularPolygon attribute), 638
 inradius (sympy.geometry.polygon.Triangle attribute), 646
 IntBaseType (class in sympy.codegen.ast), 402
 intcurve_diffequ() (in module sympy.diffgeom), 1765
 intcurve_series() (in module sympy.diffgeom), 1763
 integer, 95
 Integer (class in sympy.core.numbers), 143
 integer (sympy.assumptions.ask.AssumptionKeys attribute), 1080
 integer_elements (sympy.assumptions.ask.AssumptionKeys attribute), 1080
 Integer_nthroot() (in module sympy.core.power), 155
 IntegerPartition (class in sympy.combinatorics.partitions), 204
 IntegerRing (class in sympy.polys.domains), 912
 Integers (class in sympy.sets.fancysets), 1147
 Integral (class in sympy.integrals), 690
 Integral.is_commutative (in module sympy.integrals), 690
 integrand() (sympy.functions.special.hyper.meijerg method), 534
 integrate() (in module sympy.integrals), 688
 integrate() (sympy.core.expr.Expr method), 124
 integrate() (sympy.holonomic.holonomic.HolonomicFunction method), 659
 integrate() (sympy.polys.polyclasses.DMP method), 920
 integrate() (sympy.polys.polytools.Poly method), 839
 integrate() (sympy.series.formal.FormalPowerSeries method), 1123
 integrate_result() (in module sympy.physics.quantum.represent), 1668
 integrate_result() (in module sympy.utilities.iterables), 1423
 interior (sympy.sets.sets.Set attribute), 1133
 interior_angle (sympy.geometry.polygon.RegularPolygon attribute), 639
 interpolate() (in module sympy.polys.polyfuncs), 865
 interpolating_poly() (in module sympy.polys.specialpolys), 873
 intersect() (sympy.polys.agca.ideals.Ideal method), 894
 intersect() (sympy.polys.agca.modules.SubModule method), 890
 intersect() (sympy.sets.sets.Set method), 1134
 Intersection (class in sympy.sets.sets), 1142
 intersection() (in module sympy.geometry.util), 564
 intersection() (sympy.geometry.ellipse.Circle method), 625

[intersection\(\)](#) (sympy.geometry.ellipse.Ellipse method), [617](#)
[intersection\(\)](#) (sympy.geometry.entity.GeometryEntity method), [561](#)
[intersection\(\)](#) (sympy.geometry.line.LinearEntity method), [582](#)
[intersection\(\)](#) (sympy.geometry.plane.Plane method), [652](#)
[intersection\(\)](#) (sympy.geometry.point.Point method), [570](#)
[intersection\(\)](#) (sympy.geometry.point.Point3D method), [577](#)
[intersection\(\)](#) (sympy.geometry.polygon.Polygon method), [631](#)
[intersection\(\)](#) (sympy.sets.sets.Set method), [1134](#)
[Interval](#) (class in sympy.sets.sets), [1138](#)
[interval](#) (sympy.series.sequences.SeqBase attribute), [1116](#)
[intervals\(\)](#) (in module sympy.polys.polytools), [817](#)
[intervals\(\)](#) (sympy.polys.polyclasses.DMP method), [920](#)
[intervals\(\)](#) (sympy.polys.polytools.Poly method), [840](#)
[IntQubit](#) (class in sympy.physics.quantum.qubit), [1697](#)
[IntQubitBra](#) (class in sympy.physics.quantum.qubit), [1698](#)
[intrinsic_impedance](#) (sympy.physics.optics.medium.Medium attribute), [1714](#)
[inv\(\)](#) (sympy.matrices.matrices.MatrixBase method), [746](#)
[inv_can_transf_matrix](#) (sympy.physics.units.dimensions.DimensionSystem attribute), [1511](#)
[inv_mod\(\)](#) (sympy.matrices.matrices.MatrixBase method), [747](#)
[Inverse](#) (class in sympy.matrices.expressions), [781](#)
[inverse_ADJ\(\)](#) (sympy.matrices.matrices.MatrixBase method), [747](#)
[inverse_cosine_transform\(\)](#) (in module sympy.integrals.transforms), [672](#)
[inverse_fourier_transform\(\)](#) (in module sympy.integrals.transforms), [670](#)
[inverse_GE\(\)](#) (sympy.matrices.matrices.MatrixBase method), [747](#)
[inverse_hankel_transform\(\)](#) (in module sympy.integrals.transforms), [673](#)
[inverse_laplace_transform\(\)](#) (in module sympy.integrals.transforms), [669](#)
[inverse_LU\(\)](#) (sympy.matrices.matrices.MatrixBase method), [747](#)
[inverse_mellin_transform\(\)](#) (in module sympy.integrals.transforms), [667](#)
[inverse_sine_transform\(\)](#) (in module sympy.integrals.transforms), [671](#)
[inversion_vector\(\)](#) (sympy.combinatorics.permutations.Permutation method), [219](#)
[inversions\(\)](#) (sympy.combinatorics.permutations.Permutation method), [219](#)
[invert\(\)](#) (in module sympy.polys.polytools), [807](#)
[invert\(\)](#) (sympy.core.expr.Expr method), [124](#)
[invert\(\)](#) (sympy.polys.domains.domain.Domain method), [906](#)
[invert\(\)](#) (sympy.polys.domains.ring.Ring method), [910](#)
[invert\(\)](#) (sympy.polys.polyclasses.DMF method), [924](#)
[invert\(\)](#) (sympy.polys.polyclasses.DMP method), [920](#)
[invert\(\)](#) (sympy.polys.polytools.Poly method), [840](#)
[invert_complex\(\)](#) (in module sympy.solvers.solve), [1352](#)
[invert_real\(\)](#) (in module sympy.solvers.solve), [1352](#)
[invertible](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1080](#)
[IQFT](#) (class in sympy.physics.quantum.qft), [1696](#)
[irrational](#), [95](#)
[irrational](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1081](#)
[is_abelian](#) (sympy.combinatorics.perm_groups.PermutationGroup attribute), [244](#)
[is_above_fermi](#) (sympy.functions.special.tensor_functions.KroneckerDelta attribute), [555](#)
[is_above_fermi](#) (sympy.physics.secondquant.KroneckerDelta attribute), [1478](#)
[is_absolutely_convergent\(\)](#) (sympy.concrete.summations.Sum method), [364](#)
[is_algebraic_expr\(\)](#) (sympy.core.expr.Expr method), [124](#)
[is_aliased](#) (sympy.polys.numberfields.AlgebraicNumber attribute), [868](#)
[isalt_sym\(\)](#) (sympy.combinatorics.perm_groups.PermutationGroup method), [245](#)
[is_below_fermi](#) (sympy.functions.special.tensor_functions.KroneckerDelta attribute), [555](#)
[is_below_fermi](#) (sympy.physics.secondquant.KroneckerDelta attribute), [1479](#)

[is_negative\(\) \(sympy.polys.domains.AlgebraicField method\), 914](#)
[is_negative\(\) \(sympy.polys.domains.domain.Domain method\), 907](#)
[is_negative\(\) \(sympy.polys.domains.ExpressionDomain method\), 917](#)
[is_negative\(\) \(sympy.polys.domains.FractionField method\), 915](#)
[is_negative\(\) \(sympy.polys.domains.PolynomialRing method\), 913](#)
[is_nilpotent \(sympy.combinatorics.perm_groups.PermutationGroup attribute\), 245](#)
[is_nilpotent\(\) \(sympy.matrices.matrices.MatrixBase method\), 748](#)
[is_nonnegative\(\) \(sympy.polys.domains.AlgebraicField method\), 914](#)
[is_nonnegative\(\) \(sympy.polys.domains.domain.Domain method\), 907](#)
[is_nonnegative\(\) \(sympy.polys.domains.ExpressionDomain method\), 917](#)
[is_nonnegative\(\) \(sympy.polys.domains.FractionField method\), 915](#)
[is_nonnegative\(\) \(sympy.polys.domains.PolynomialRing method\), 913](#)
[is_nonpositive\(\) \(sympy.polys.domains.AlgebraicField method\), 914](#)
[is_nonpositive\(\) \(sympy.polys.domains.domain.Domain method\), 907](#)
[is_nonpositive\(\) \(sympy.polys.domains.ExpressionDomain method\), 917](#)
[is_nonpositive\(\) \(sympy.polys.domains.FractionField method\), 915](#)
[is_nonpositive\(\) \(sympy.polys.domains.PolynomialRing method\), 913](#)
[is_nonzero \(sympy.geometry.point.Point attribute\), 571](#)
[is_normal\(\) \(sympy.combinatorics.perm_groups.PermutationGroup method\), 246](#)
[is_normalized \(sympy.physics.quantum.state.Wavefunction attribute\), 1686](#)
[is_nthpow_residue\(\) \(in module sympy.ntheory.residue_ntheory\), 328](#)
[is_number \(sympy.core.expr.Expr attribute\), 126](#)
[is_number \(sympy.functions.special.hyper.meijerg attribute\), 534](#)
[is_odd \(sympy.combinatorics.permutations.Permutation attribute\), 221](#)
[is_one \(sympy.polys.polyclasses.ANP attribute\), 924](#)
[is_one \(sympy.polys.polyclasses.DMF attribute\), 924](#)
[is_done \(sympy.polys.polyclasses.DMP attribute\), 921](#)
[is_one \(sympy.polys.polytools.Poly attribute\), 843](#)
[is_domain \(sympy.polys.domains.domain.Domain method\), 907](#)
[is_only_above_fermi \(sympy.functions.special.tensor_functions.KroneckerDelta attribute\), 556](#)
[is_only_above_fermi \(sympy.physics.secondquant.KroneckerDelta attribute\), 1479](#)
[is_only_below_fermi \(sympy.functions.special.tensor_functions.KroneckerDelta attribute\), 556](#)
[is_only_below_fermi \(sympy.physics.secondquant.KroneckerDelta attribute\), 1480](#)
[is_ordinaryannihilator \(sympy.physics.secondquant.AnnihilateFermion attribute\), 1482](#)
[is_only_q annihilator \(sympy.physics.secondquant.CreateFermion attribute\), 1484](#)
[is_fieldy_q_creator \(sympy.physics.secondquant.AnnihilateFermion attribute\), 1482](#)
[is_domain_q_creator \(sympy.physics.secondquant.CreateFermion attribute\), 1484](#)
[is_operator \(sympy.sets.sets.Set attribute\), 1134](#)
[is_parallel\(\) \(sympy.geometry.line.LinearEntity method\), 583](#)
[is_parallel\(\) \(sympy.geometry.plane.Plane method\), 653](#)
[is_perpendicular\(\) \(sympy.geometry.line.LinearEntity method\), 584](#)
[is_perpendicular\(\) \(sympy.geometry.plane.Plane method\), 583](#)
[is_polynomial\(\) \(sympy.core.expr.Expr method\), 127](#)
[is_positive\(\) \(sympy.polys.domains.AlgebraicField method\), 914](#)
[is_positive\(\) \(sympy.polys.domains.domain.Domain method\), 907](#)
[is_positive\(\) \(sympy.polys.domains.ExpressionDomain method\), 917](#)
[is_positive\(\) \(sympy.polys.domains.FractionField method\), 916](#)
[is_positive\(\) \(sympy.polys.domains.PolynomialRing method\), 913](#)
[is_primary\(\) \(sympy.polys.agca.ideals.Ideal method\), 894](#)
[is_prime\(\) \(sympy.polys.agca.ideals.Ideal method\), 894](#)

`is_primitive` (sympy.polys.polyclasses.DMP attribute), 921

`is_primitive` (sympy.polys.polytools.Poly attribute), 843

`is_primitive()` (sympy.combinatorics.perm_groups.PermutationGroup method), 246

`is_primitive_root()` (in module sympy.ntheory.residue_ntheory), 327

`is_principal()` (sympy.polys.agca.ideals.Ideal method), 894

`is_proper_subset()` (sympy.sets.sets.Set method), 1135

`is_proper_superset()` (sympy.sets.sets.Set method), 1135

`is_q_annihilator` (sympy.physics.secondquant.AnnihilationOperator attribute), 1483

`is_q_annihilator` (sympy.physics.secondquant.CreationOperator attribute), 1484

`is_q_creator` (sympy.physics.secondquant.AnnihilationOperator attribute), 1483

`is_q_creator` (sympy.physics.secondquant.CreationOperator attribute), 1485

`is_quad_residue()` (in module sympy.ntheory.residue_ntheory), 328

`is_quadratic` (sympy.polys.polyclasses.DMP attribute), 921

`is_quadratic` (sympy.polys.polytools.Poly attribute), 844

`is_radical()` (sympy.polys.agca.ideals.Ideal method), 894

`is_rational_function()` (sympy.core.expr.Expr method), 128

`is_reduced()` (in module sympy.polys.groebnertools), 997

`is_right()` (sympy.geometry.polygon.Triangle method), 647

`is_right_unbounded` (sympy.sets.sets.Interval attribute), 1139

`is_scalar_multiple()` (sympy.geometry.point.Point method), 571

`is_scalene()` (sympy.geometry.polygon.Triangle method), 647

`is_sequence()` (in module sympy.core.compatibility), 199

`is_similar()` (sympy.geometry.entity.GeometryEntity method), 562

`is_similar()` (sympy.geometry.line.LinearEntity method), 584

`is_similar()` (sympy.geometry.polygon.Triangle method), 647

`is_simple()` (sympy.functions.special.delta_functions.DiracDelta method), 478

`is_Singleton` (sympy.combinatorics.permutations.Permutation attribute), 221

`is_solenoidal()` (in module sympy.physics.vector.fieldfunctions), 1542

`is_solenoidal()` (in module sympy.vector), 1805

`is_solvable` (sympy.combinatorics.perm_groups.PermutationGroup attribute), 246

`is_sqf` (sympy.polys.polyclasses.DMP attribute), 921

`is_sqf` (sympy.polys.polytools.Poly attribute), 844

`is_strictly_decreasing()` (in module sympy.calculus.singularities), 1462

`is_strictly_increasing()` (in module sympy.calculus.singularities), 1462

`is_subalgebra()` (sympy.categories.Diagram method), 1743

`is_subgroup()` (sympy.combinatorics.perm_groups.PermutationGroup method), 247

`is_submodule()` (sympy.polys.agca.modules.FreeModule method), 888

`is_submodule()` (sympy.polys.agca.modules.Module method), 887

`is_submodule()` (sympy.polys.agca.modules.QuotientModule method), 896

`is_submodule()` (sympy.polys.agca.modules.SubModule method), 891

`is_subset()` (sympy.sets.sets.Set method), 1135

`is_superset()` (sympy.sets.sets.Set method), 1135

`is_surjective()` (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 901

`is_tangent()` (sympy.geometry.ellipse.Ellipse method), 618

`is_transitive()` (sympy.combinatorics.perm_groups.PermutationGroup method), 247

`is_trivial` (sympy.combinatorics.perm_groups.PermutationGroup attribute), 248

`is_true` (sympy.assumptions.ask.AssumptionKeys attribute), 1081

`is_univariate` (sympy.polys.polytools.Poly attribute), 844

`is_whole_ring()` (sympy.polys.agca.ideals.Ideal method), 894

`is_zero` (sympy.geometry.point.Point attribute), 571

`is_zero` (sympy.matrices.immutable.ImmutableDenseMatrix attribute), 777

`is_zero` (sympy.polys.polyclasses.ANP attribute), 925

is_zero (sympy.polys.polyclasses.DMF attribute), 924
 is_zero (sympy.polys.polyclasses.DMP attribute), 921
 is_zero (sympy.polys.polytools.Poly attribute), 844
 is_zero() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 902
 is_zero() (sympy.polys.agca.ideals.Ideal method), 894
 is_zero() (sympy.polys.agca.modules.FreeModule method), 888
 is_zero() (sympy.polys.agca.modules.Module method), 887
 is_zero() (sympy.polys.agca.modules.QuotientModule method), 896
 is_zero() (sympy.polys.agca.modules.SubModule method), 891
 is_zero() (sympy.polys.domains.domain.Domain method), 907
 is_zero_dimensional (sympy.polys.polytools.GroebnerBasis attribute), 863
 is_zero_dimensional() (in module sympy.polys.polytools), 821
 isdisjoint() (sympy.sets.sets.Set method), 1135
 isgen (class in sympy.codegen.fnodes), 419
 isolate() (in module sympy.polys.numberfields), 867
 IsomorphismFailed (class in sympy.polys.polyerrors), 999
 isprime() (in module sympy.ntheory.primetest), 326
 issubset() (sympy.sets.sets.Set method), 1136
 issuperset() (sympy.sets.sets.Set method), 1136
 ITE (class in sympy.logic.boolalg), 725
 items() (sympy.core.containers.Dict method), 198
 iter_q_annihilators() (sympy.physics.secondquant.NO method), 1492
 iter_q_creators() (sympy.physics.secondquant.NO method), 1492
 iterable() (in module sympy.core.compatibility), 198
 iterate_binary() (sympy.combinatorics.subsets.Subset method), 268
 iterate_graycode() (sympy.combinatorics.subsets.Subset method), 269
 itercoeffs() (sympy.polys.rings.PolyElement method), 977
 itermonomials() (in module sympy.polys.monomials), 868
 itermonoms() (sympy.polys.rings.PolyElement method), 977
 iterterms() (sympy.polys.rings.PolyElement method), 977
 jacobi (class in sympy.functions.special.polynomials), 540
 jacobi_normalized() (in module sympy.functions.special.polynomials), 541
 jacobi_poly() (in module sympy.polys.orthopolys), 874
 jacobi_symbol() (in module sympy.ntheory.residue_ntheory), 329
 jacobian() (sympy.diffgeom.CoordSystem method), 1756
 JavascriptCodePrinter (class in sympy.printing.jscode), 1035
 jn (class in sympy.functions.special.bessel), 515
 jn_zeros() (in module sympy.functions.special.bessel), 517
 join() (sympy.physics.continuum_mechanics.beam.Beam method), 1727
 jordan_cell() (in module sympy.matrices.dense), 756
 josephus() (sympy.combinatorics.permutations.Permutation class method), 221
 jscode() (in module sympy.printing.jscode), 1035
 julia_code() (in module sympy.printing.julia), 1037
 JuliaCodeGen (class in sympy.utilities.codegen), 1403
 JuliaCodePrinter (class in sympy.printing.julia), 1037
 JxBra (class in sympy.physics.quantum.spin), 1673
 JxBraCoupled (class in sympy.physics.quantum.spin), 1676
 JxKet (class in sympy.physics.quantum.spin), 1673
 JxKetCoupled (class in sympy.physics.quantum.spin), 1676
 JyBra (class in sympy.physics.quantum.spin), 1674
 JyBraCoupled (class in sympy.physics.quantum.spin), 1676

JyKet (class in `sympy.physics.quantum.spin`), 1674
 JyKetCoupled (class in `sympy.physics.quantum.spin`), 1676
 JzBra (class in `sympy.physics.quantum.spin`), 1676
 JzBraCoupled (class in `sympy.physics.quantum.spin`), 1678
 JzKet (class in `sympy.physics.quantum.spin`), 1674
 JzKetCoupled (class in `sympy.physics.quantum.spin`), 1676
K
 kahane_simplify() (in module `sympy.physics.hep.gamma_matrices`), 1515
 kanes_equations() (`sympy.physics.mechanics.kane.KanesMethod` method), 1633
 KanesMethod (class in `sympy.physics.mechanics.kane`), 1631
 kbins() (in module `sympy.utilities.iterables`), 1423
 kernel() (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 902
 Ket (class in `sympy.physics.quantum.state`), 1681
 ket (`sympy.physics.quantum.operator.OuterProduct` attribute), 1661
 ket (`sympy.physics.secondquant.InnerProduct` attribute), 1486
 KetBase (class in `sympy.physics.quantum.state`), 1680
 key2bounds() (`sympy.matrices.matrices.MatrixBase` method), 748
 key2ij() (`sympy.matrices.matrices.MatrixBase` method), 748
 keys() (`sympy.core.containers.Dict` method), 198
 kid_rsa_private_key() (in module `sympy.crypto.crypto`), 350
 kid_rsa_public_key() (in module `sympy.crypto.crypto`), 350
 killable_index (`sympy.functions.special.tensor_functions.KroneckerDelta` attribute), 556
 killable_index (`sympy.physics.secondquant.KroneckerDelta` attribute), 1480
 kin_explicit_rhs (`sympy.physics.mechanics.system.SymbolicSystem` attribute), 1641
 kind (class in `sympy.codegen.fnodes`), 419
 kindiffdict() (`sympy.physics.mechanics.kane.KanesMethod` method), 1633
 kinematic_equations() (in module `sympy.physics.vector.functions`), 1569
 kinetic_energy() (in module `sympy.physics.mechanics.functions`), 1629
 kinetic_energy() (`sympy.physics.mechanics.particle.Particle` method), 1622
 kinetic_energy() (`sympy.physics.mechanics.rigidbody.RigidBody` method), 1624
 known_fcns_src1 (in module `sympy.printing.julia`), 1037
 known_fcns_src1 (in module `sympy.printing.octave`), 1039
 known_fcns_src2 (in module `sympy.printing.julia`), 1037
 known_fcns_src2 (in module `sympy.printing.octave`), 1039
 known_functions (in module `sympy.printing.jscode`), 1035
 known_functions (in module `sympy.printing.mathematica`), 1034
 known_functions (in module `sympy.printing.rcode`), 1028
 known_functions (in module `sympy.printing.rust`), 1042
 known_functions_C89 (in module `sympy.printing.ccode`), 1024
 known_functions_C99 (in module `sympy.printing.ccode`), 1024
 KroneckerDelta (class in `sympy.functions.special.tensor_functions`), 553
 KroneckerDelta (class in `sympy.physics.secondquant`), 1477
 ksubsets() (`sympy.combinatorics.subsets` method), 274
 Kumaraswamy() (in module `sympy.stats`), 1206
 kwargs() (`sympy.codegen.ast.Token` method), 404
L
 l1_norm() (`sympy.polys.polyclasses.DMP` method), 920
 l1_norm() (`sympy.polys.polytools.Poly` method), 845
 L2 (class in `sympy.physics.quantum.hilbert`), 1657
 Label (class in `sympy.codegen.cnodes`), 413
 label (`sympy.physics.quantum.state.TimeDepState` attribute), 1683
 label (`sympy.tensor.indexed.Idx` attribute), 1368

- label (sympy.tensor.indexed.IndexedBase attribute), 1371
- labeller() (in module sympy.physics.quantum.circuitplot), 1688
- LagrangesMethod (class in sympy.physics.mechanics.lagrange), 1634
- Lagrangian() (in module sympy.physics.mechanics.functions), 1630
- laguerre (class in sympy.functions.special.polynomials), 548
- laguerre_poly() (in module sympy.polys.orthopolys), 875
- Lambda (class in sympy.core.function), 176
- lambda_notation() (in module sympy.parsing.sympy_parser), 1458
- LambdaPrinter (class in sympy.printing.lambdarepr), 1045
- lambdarepr() (in module sympy.printing.lambdarepr), 1045
- lambdastr() (in module sympy.utilities.lambdify), 1438
- lambdify() (in module sympy.utilities.lambdify), 1438
- LambertW (class in sympy.functions.elementary.exponential), 449
- Laplace() (in module sympy.stats), 1207
- laplace_transform() (in module sympy.integrals.transforms), 668
- latex() (in module sympy.printing.latex), 1045
- LaTeXParsingError (class in sympy.parsing.latex), 1459
- LatexPrinter (class in sympy.printing.latex), 1045
- lbound() (in module sympy.codegen.fnodes), 419
- LC() (in module sympy.polys.polytools), 804
- LC() (sympy.polys.polyclasses.ANP method), 924
- LC() (sympy.polys.polyclasses.DMP method), 918
- LC() (sympy.polys.polytools.Poly method), 822
- lcm() (in module sympy.polys.polytools), 812
- lcm() (sympy.core.numbers.Number method), 138
- lcm() (sympy.polys.domains.domain.Domain method), 907
- lcm() (sympy.polys.domains.field.Field method), 909
- lcm() (sympy.polys.domains.PolynomialRing method), 913
- lcm() (sympy.polys.domains.RealField method), 916
- lcm() (sympy.polys.polyclasses.DMP method), 921
- lcm() (sympy.polys.polytools.Poly method), 845
- lcm_list() (in module sympy.polys.polytools), 813
- ldescent() (in module sympy.solvers.diophantine), 1336
- LDLdecomposition() (sympy.matrices.matrices.MatrixBase method), 739
- LDLdecomposition() (sympy.matrices.sparse.SparseMatrix method), 768
- LDLsolve() (sympy.matrices.matrices.MatrixBase method), 739
- Le (in module sympy.core.relational), 161
- leading_expv() (sympy.polys.rings.PolyElement method), 977
- leading_monom() (sympy.polys.rings.PolyElement method), 977
- leading_term() (sympy.polys.rings.PolyElement method), 978
- leadterm() (sympy.core.expr.Expr method), 129
- left (sympy.sets.sets.Interval attribute), 1139
- left() (sympy.printing.pretty.stringpict.stringPict method), 1056
- left_open (sympy.sets.sets.Interval attribute), 1140
- leftslash() (sympy.printing.pretty.stringpict.stringPict method), 1056
- legendre (class in sympy.functions.special.polynomials), 546
- legendre_poly() (in module sympy.polys.orthopolys), 874
- legendre_symbol() (in module sympy.ntheory.residue_ntheory), 329
- length (sympy.geometry.curve.Curve attribute), 610
- length (sympy.geometry.line.LinearEntity attribute), 585
- length (sympy.geometry.line.Segment attribute), 596
- length (sympy.geometry.point.Point attribute), 571
- length (sympy.geometry.polygon.RegularPolygon attribute), 639

`length` (sympy.physics.continuum_mechanics.beam.Beam attribute), 1728

`length` (sympy.series.sequences.SeqBase attribute), 1116

`length()` (sympy.combinatorics.permutations.Permutation method), 222

`length()` (sympy.polys.polytools.Poly method), 845

`lens_formula()` (in module sympy.physics.optics.utils), 1718

`lens_makers_formula()` (in module sympy.physics.optics.utils), 1717

`lerchphi` (class in sympy.functions.special.zeta_functions), 527

`LessThan` (class in sympy.core.relational), 166

`LeviCivita` (class in sympy.functions.special.tensor_functions), 553

`LexOrder` (class in sympy.polys.orderings), 869

`lfsr_autocorrelation()` (in module sympy.crypto.crypto), 353

`lfsr_connection_polynomial()` (in module sympy.crypto.crypto), 353

`lfsr_sequence()` (in module sympy.crypto.crypto), 352

`Li` (class in sympy.functions.special.error_functions), 506

`li` (class in sympy.functions.special.error_functions), 504

`lie_algebra()` (sympy.liealgebras.type_a.TypeA method), 706

`lie_algebra()` (sympy.liealgebras.type_b.TypeB method), 707

`lie_algebra()` (sympy.liealgebras.type_c.TypeC method), 708

`lie_algebra()` (sympy.liealgebras.type_d.TypeD method), 710

`lie_heuristic_abaco1_product()` (in module sympy.solvers.ode), 1262

`lie_heuristic_abaco1_simple()` (in module sympy.solvers.ode), 1261

`lie_heuristic_abaco2_similar()` (in module sympy.solvers.ode), 1263

`lie_heuristic_abaco2_unique_general()` (in module sympy.solvers.ode), 1265

`lie_heuristic_abaco2_unique_unknown()` (in module sympy.solvers.ode), 1264

`lie_heuristic_bivariate()` (in module sympy.solvers.ode), 1262

`lie_heuristic_chi()` (in module sympy.solvers.ode), 1263

`lie_heuristic_function_sum()` (in module sympy.solvers.ode), 1264

`lie_heuristic_linear()` (in module sympy.solvers.ode), 1265

`lie_derivative` (class in sympy.diffgeom), 1762

`lift()` (sympy.polys.polyclasses.DMP method), 921

`lift()` (sympy.polys.polytools.Poly method), 845

`Limit` (class in sympy.series.limits), 1106

`limit()` (in module sympy.series.limits), 1105

`limit()` (sympy.core.expr.Expr method), 129

`limit_denominator()` (sympy.core.numbers.Rational method), 143

`limit_seq()` (in module sympy.series.limitseq), 1131

`limitinf()` (in module sympy.series.gruntz), 1108

`limits` (sympy.geometry.curve.Curve attribute), 610

`limits` (sympy.physics.quantum.state.Wavefunction attribute), 1686

`Line` (class in sympy.geometry.line), 589

`Line2D` (class in sympy.geometry.line), 599

`Line2DBaseSeries` (class in sympy.plotting.plot), 1071

`Line3D` (class in sympy.geometry.line), 604

`Line3DBaseSeries` (class in sympy.plotting.plot), 1071

`line_integrate()` (in module sympy.integrals), 690

`linear_eq_to_matrix()` (in module sympy.solvers.solve), 1353

`linear_momentum()` (in module sympy.physics.mechanics.functions), 1627

`linear_momentum()` (sympy.physics.mechanics.particle.Particle method), 1622

`linear_momentum()` (sympy.physics.mechanics.rigidbody.RigidBody method), 1625

`LinearEntity` (class in sympy.geometry.line), 579

`LinearEntity2D` (class in sympy.geometry.line), 597

`LinearEntity3D` (class in sympy.geometry.line), 603

`linearize()` (sympy.physics.mechanics.kane.KanesMethod method), 1633

`linearize()` (sympy.physics.mechanics.lagrange.LagrangesMethod method), 1636

linearize() (sympy.physics.mechanics.linearize.linearize method), 1642

Linearizer (class in sympy.physics.mechanics.linearize), 1641

LineOver1DRangeSeries (class in sympy.plotting.plot), 1071

linsolve() (in module sympy.solvers.solve), 1354

list() (sympy.combinatorics.permutations.Cycle method), 231

list() (sympy.combinatorics.permutations.Permutation method), 222

list2numpy() (in module sympy.matrices.dense), 759

list_can_dims (sympy.physics.units.dimensions.Dimension attribute), 1511

list_visitor() (in module sympy.utilities.enumerative), 1412

listcoeffs() (sympy.polys.rings.PolyElement method), 978

listmonoms() (sympy.polys.rings.PolyElement method), 978

listterms() (sympy.polys.rings.PolyElement method), 978

literal_dp (class in sympy.codegen.fnodes), 419

literal_sp (class in sympy.codegen.fnodes), 419

liupc() (sympy.matrices.sparse.SparseMatrix method), 769

LM() (in module sympy.polys.polytools), 804

LM() (sympy.polys.polytools.Poly method), 823

load (sympy.physics.continuum_mechanics.beam.Beam attribute), 1728

loads (sympy.physics.mechanics.system.SymbolicSystem attribute), 1641

locate_new() (sympy.vector.coordsysrect.CoordSys3D method), 1787

locatenew() (sympy.physics.vector.point.Point method), 1565

log (class in sympy.functions.elementary.exponential), 450

log() (sympy.polys.domains.domain.Domain method), 907

log() (sympy.polys.domains.IntegerRing method), 912

log10 (class in sympy.codegen.cfunctions), 412

log1p (class in sympy.codegen.cfunctions), 412

log2 (class in sympy.codegen.cfunctions), 412

loggamma (class in sympy.functions.special.gamma_functions), 484

Logistic() (in module sympy.stats), 1207

LogNormal() (in module sympy.stats), 1208

Lopen() (sympy.sets.sets.Interval class method), 1139

lower (sympy.physics.secondquant.AntiSymmetricTensor attribute), 1494

lower (sympy.tensor.indexed.Idx attribute), 1368

lower_central_series() (sympy.combinatorics.perm_groups.PermutationGroup method), 248

lower_triangular (sympy.assumptions.ask.AssumptionKeys attribute), 1081

lower_triangular_solve() (sympy.matrices.matrices.MatrixBase method), 748

lowergamma (class in sympy.functions.special.gamma_functions), 489

lseries() (sympy.core.expr.Expr method), 129

Lt (in module sympy.core.relational), 161

LT() (in module sympy.polys.polytools), 804

LT() (sympy.polys.polytools.Poly method), 823

ltrim() (sympy.polys.polytools.Poly method), 846

lucas (class in sympy.functions.combinatorial.numbers), 471

LUdecomposition() (sympy.matrices.matrices.MatrixBase method), 740

LUdecomposition_Simple() (sympy.matrices.matrices.MatrixBase method), 741

LUdecompositionFF() (sympy.matrices.matrices.MatrixBase method), 740

LUsolve() (sympy.matrices.matrices.MatrixBase method), 741

M

magnitude() (sympy.physics.vector.vector.Vector method), 1559

magnitude() (sympy.vector.vector.Vector method), 1794

major (sympy.geometry.ellipse.Ellipse attribute), 618

make_perm() (sympy.combinatorics.perm_groups.Permutation method), 249

make_routine() (in module sympy.utilities.codegen), 1407

Manifold (class in `sympy.diffgeom`), 1753
map() (`sympy.polys.domains.domain.Domain` method), 907
mass (`sympy.physics.mechanics.particle.Particle` attribute), 1623
mass_matrix (`sympy.physics.mechanics.kane.KanesMethod` attribute), 1634
mass_matrix (`sympy.physics.mechanics.lagrange.LagrangeMethod` attribute), 1636
mass_matrix_full (`sympy.physics.mechanics.kane.KanesMethod` attribute), 1634
mass_matrix_full (`sympy.physics.mechanics.lagrange.LagrangeMethod` attribute), 1636
MatAdd (class in `sympy.matrices.expressions`), 781
match() (`sympy.core.basic.Basic` method), 103
matches() (`sympy.core.basic.Basic` method), 103
mathematica() (in module `sympy.parsing.mathematica`), 1456
mathematica_code() (in module `sympy.printing.mathematica`), 1034
MathieuBase (class in `sympy.functions.special.mathieu_functions`), 537
mathieuc (class in `sympy.functions.special.mathieu_functions`), 538
mathieucprime (class in `sympy.functions.special.mathieu_functions`), 539
mathieus (class in `sympy.functions.special.mathieu_functions`), 537
mathieusprime (class in `sympy.functions.special.mathieu_functions`), 538
mathml() (in module `sympy.printing.mathml`), 1048
mathml_tag() (`sympy.printing.mathml.MathMLContentPrinter` method), 1048
mathml_tag() (`sympy.printing.mathml.MathMLPresentationPrinter` method), 1048
MathMLContentPrinter (class in `sympy.printing.mathml`), 1048
MathMLPresentationPrinter (class in `sympy.printing.mathml`), 1048
MathMLPrinterBase (class in `sympy.printing.mathml`), 1047
MatMul (class in `sympy.matrices.expressions`), 781
MatPow (class in `sympy.matrices.expressions`), 781
matrix2numpy() (in module `sympy.matrices.dense`), 759
matrix_fglm() (in module `sympy.polys.fglmtools`), 997
matrix_form() (`sympy.liealgebras.weyl_group.WeylGroup` method), 716
matrix_multiply_elementwise() (in module `sympy.matrices.dense`), 754
matrix_rep() (in module `sympy.physics.secondquant`), 1489
matrix_to_density() (in module `sympy.physics.quantum.qubit`), 1699
matrix_to_qubit() (in module `sympy.physics.quantum.qubit`), 1698
matrix_to_vector() (in module `sympy.vector`), 1801
MatrixBase (class in `sympy.matrices.matrices`), 738
MatrixError (class in `sympy.matrices.matrices`), 754
MatrixExpr (class in `sympy.matrices.expressions`), 778
MatrixSymbol (class in `sympy.matrices.expressions`), 780
Max (class in `sympy.functions.elementary.miscellaneous`), 453
max (in `sympy.codegen.ast.FloatType` attribute), 400
max() (`sympy.combinatorics.permutations.Permutation` method), 223
max_bmoment() (`sympy.physics.continuum_mechanics.beam.Beam` method), 1728
max_deflection() (`sympy.physics.continuum_mechanics.beam.Beam` method), 1728
max_div (in `sympy.combinatorics.perm_groups.PermutationGroup` attribute), 249
max_exponent (in `sympy.codegen.ast.FloatType` attribute), 400
max_norm() (`sympy.polys.polyclasses.DMP` method), 921
max_order() (`sympy.polys.polytools.Poly` method), 846
Maxwell() (in module `sympy.stats`), 1209
MCodePrinter (class in `sympy.printing.mathematica`), 1034
mdft() (in module `sympy.physics.matrices`), 1472
measure (`sympy.sets.sets.Set` attribute), 1136
measure_all() (in module `sympy.physics.quantum.qubit`), 1699
measure_all_oneshot() (in module `sympy.physics.quantum.qubit`), 1700

measure_partial() (in module sympy.physics.quantum.qubit), 1699
 measure_partial_oneshot() (in module sympy.physics.quantum.qubit), 1700
 medial (sympy.geometry.polygon.Triangle attribute), 648
 medians (sympy.geometry.polygon.Triangle attribute), 648
 Medium (class in sympy.physics.optics.medium), 1713
 meets() (sympy.series.gruntz.SubSet method), 1110
 meijerg (class in sympy.functions.special.hyper), 532
 mellin_transform() (in module sympy.integrals.transforms), 667
 memoize_property() (in module sympy.utilities.decorator), 1409
 merge (class in sympy.codegen.fnodes), 419
 merge_solution() (in module sympy.solvers.diophantine), 1334
 metric_to_Christoffel_1st() (in module sympy.diffgeom), 1766
 metric_to_Christoffel_2nd() (in module sympy.diffgeom), 1767
 metric_to_Ricci_components() (in module sympy.diffgeom), 1767
 metric_to_Riemann_components() (in module sympy.diffgeom), 1767
 mgamma() (in module sympy.physics.matrices), 1473
 midpoint (sympy.geometry.line.Segment attribute), 596
 midpoint() (sympy.geometry.point.Point method), 571
 Min (class in sympy.functions.elementary.miscellaneous), 452
 min() (sympy.combinatorics.permutations.Permutation method), 223
 min_exponent (sympy.codegen.ast.FloatType attribute), 400
 min_qubits (sympy.physics.quantum.gate.CGate attribute), 1690
 min_qubits (sympy.physics.quantum.gate.CNotGate attribute), 1692
 min_qubits (sympy.physics.quantum.gate.Gate attribute), 1689
 minimal_block() (sympy.combinatorics.perm_groups.PermGroup method), 250
 minimal_blocks() (sympy.combinatorics.perm_groups.PermGroup method), 250
 minimal_polynomial() (in module sympy.polys.numberfields), 866
 minlex() (in module sympy.utilities.iterables), 1424
 minor (sympy.geometry.ellipse.Ellipse attribute), 619
 minpoly() (in module sympy.polys.numberfields), 866
 mirror_formula() (in module sympy.physics.optics.utils), 1718
 mobius (class in sympy.ntheory), 333
 mobiusrange() (sympy.ntheory.generate.Sieve method), 299
 Mod (class in sympy.core.mod), 161
 modgcd_bivariate() (in module sympy.polys.modulargcd), 1001
 modgcd_multivariate() (in module sympy.polys.modulargcd), 1002
 modgcd_univariate() (in module sympy.polys.modulargcd), 1000
 Module (class in sympy.codegen.fnodes), 416
 Module (class in sympy.polys.agca.modules), 886
 module_quotient() (sympy.polys.agca.modules.SubModule method), 891
 ModuleHomomorphism (class in sympy.polys.agca.homomorphisms), 900
 momentum (sympy.physics.quantum.cartesian.PxBra attribute), 1656
 momentum (sympy.physics.quantum.cartesian.PxKet attribute), 1656
 monic() (in module sympy.polys.polytools), 813
 monic() (sympy.polys.polyclasses.DMP method), 921
 monic() (sympy.polys.polytools.Poly method), 846
 monic() (sympy.polys.rings.PolyElement method), 978
 Monomial (class in sympy.polys.monomials), 868
 monomial_basis() (sympy.polys.rings.PolyRing method), 974
 monomial_count() (in module sympy.polys.monomials), 868
 monoms() (sympy.polys.polyclasses.DMP method), 921
 monoms() (sympy.polys.polytools.Poly method), 846
 monoms() (sympy.polys.rings.PolyElement method), 978
 monotonicity_helper() (in module sympy.calculus.singularities), 1463
 Morphism (class in sympy.categories), 1736

- morphisms (sympy.categories.diagram_drawing module attribute), 1747
- MpmathPrinter (class in sympy.printing.pycode), 1048
- mr() (in module sympy.ntheory.primetest), 325
- mr() (in module sympy.series.gruntz), 1109
- mr_leadterm() (in module sympy.series.gruntz), 1108
- mr_max1() (in module sympy.series.gruntz), 1109
- mr_max3() (in module sympy.series.gruntz), 1109
- msigma() (in module sympy.physics.matrices), 1473
- msubs() (in module sympy.physics.mechanics), 1642
- Mul (class in sympy.core.mul), 155
- Mul() (sympy.assumptions.handlers.calculus.AskFiniteAndNotFinite), 1096
- Mul() (sympy.assumptions.handlers.sets.AskAntisymmetric), 1098
- Mul() (sympy.assumptions.handlers.sets.AskHermitian), 1098
- Mul() (sympy.assumptions.handlers.sets.AskImaginary), 1099
- Mul() (sympy.assumptions.handlers.sets.AskInteger), 1099
- Mul() (sympy.assumptions.handlers.sets.AskRational), 1099
- Mul() (sympy.assumptions.handlers.sets.AskReal), 1099
- mul() (sympy.polys.domains.domain.Domain method), 907
- mul() (sympy.polys.polyclasses.DMF method), 924
- mul() (sympy.polys.polyclasses.DMP method), 921
- mul() (sympy.polys.polytools.Poly method), 847
- mul() (sympy.polys.rings.PolyRing method), 974
- mul_ground() (sympy.polys.polyclasses.DMP method), 921
- mul_ground() (sympy.polys.polytools.Poly method), 847
- mul_inv() (sympy.combinatorics.permutations.Permutation method), 223
- mul_xin() (in module sympy.polys.ring_series), 1019
- MultiFactorial (class in sympy.functions.combinatorial.factorials), 472
- multinomial_coefficients() (in module sympy.ntheory.multinomial), 324
- multinomial_coefficients_iterator() (in module sympy.ntheory.multinomial), 325
- multiplicity() (in module sympy.ntheory.factor_), 308
- multiply() (sympy.matrices.matrices.MatrixBase method), 748
- multiply_ideal() (sympy.polys.agca.modules.FreeModule method), 889
- multiply_ideal() (sympy.polys.agca.modules.Module method), 887
- multiply_ideal() (sympy.polys.agca.modules.SubModule method), 892
- multiset() (in module sympy.utilities.iterables), 1425
- multiset_combinations() (in module sympy.utilities.iterables), 1425
- multiset_partitions() (in module sympy.utilities.iterables), 1425
- multiset_partitions_taocp() (in module sympy.utilities.enumerative), 1410
- multiset_partitions() (in module sympy.utilities.iterables), 1427
- MultisetHandler (class in sympy.utilities.enumerative), 1412
- MutableDensePolynomialError (class in sympy.polys.polyerrors), 1000
- MutableDenseMatrix (class in sympy.matrices.dense), 762
- MutableDenseNDimArray (class in sympy.tensor.array), 1363
- MutableSparseMatrix (class in sympy.matrices.sparse), 772
- MutableSparseNDimArray (class in sympy.tensor.array), 1363
- Mx (class in sympy.physics.quantum.circuitplot), 1689
- Mz (class in sympy.physics.quantum.circuitplot), 1689
- ## N
- n (sympy.combinatorics.graycode.GrayCode attribute), 275
- N (sympy.physics.quantum.shor.CMod attribute), 1701
- N (in module sympy.core.evalf), 197
- n() (sympy.geometry.point.Point method), 572
- n() (sympy.polys.domains.domain.Domain method), 907
- n_order() (in module sympy.ntheory.residue_ntheory), 326
- Nakagami() (in module sympy.stats), 1210

- name (sympy.categories.Category attribute), 1741
- name (sympy.categories.NamedMorphism attribute), 1738
- NamedMorphism (class in sympy.categories), 1737
- NaN (class in sympy.core.numbers), 146
- Nand (class in sympy.logic.boolalg), 723
- nargs (sympy.core.function.FunctionClass attribute), 183
- native_coeffs() (sympy.polys.numberfields.AlgebraicNumber method), 868
- Naturals (class in sympy.sets.fancysets), 1146
- Naturals0 (class in sympy.sets.fancysets), 1147
- Ne (in module sympy.core.relational), 161
- necklaces() (in module sympy.utilities.iterables), 1427
- neg() (sympy.polys.domains.domain.Domain method), 907
- neg() (sympy.polys.polyclasses.DMF method), 924
- neg() (sympy.polys.polyclasses.DMP method), 921
- neg() (sympy.polys.polytools.Poly method), 847
- negative, 95
- negative (sympy.assumptions.ask.AssumptionKeys attribute), 1082
- NegativeInfinity (class in sympy.core.numbers), 148
- NegativeOne (class in sympy.core.numbers), 145
- new() (sympy.polys.polytools.Poly class method), 848
- newtons_method() (in module sympy.codegen.algorithms), 421
- newtons_method_function() (in module sympy.codegen.algorithms), 422
- next() (sympy.combinatorics.graycode.GrayCode method), 276
- next() (sympy.combinatorics.prufer.Prufer method), 264
- next() (sympy.printing.pretty.stringpict.StringPicture static method), 1056
- next_binary() (sympy.combinatorics.subsets.Subset method), 269
- next_gray() (sympy.combinatorics.subsets.Subset method), 269
- next_lex() (sympy.combinatorics.partitions.IntegerPartitions method), 205
- next_lex() (sympy.combinatorics.permutations.Permutation method), 223
- next_lexicographic() (sympy.combinatorics.subsets.Subset method), 270
- next_nonlex() (sympy.combinatorics.permutations.Permutation method), 224
- next_trotterjohnson() (sympy.combinatorics.permutations.Permutation method), 224
- nextprime() (in module sympy.ntheory.generate), 302
- nth_root() (in module sympy.core.function), 196
- nine_point_circle (sympy.geometry.polygon.Triangle attribute), 648
- nnz() (sympy.matrices.sparse.SparseMatrix method), 770
- NO (class in sympy.physics.secondquant), 1490
- no_attrs_in_subclass (class in sympy.utilities.decorator), 1409
- Node (class in sympy.codegen.ast), 402
- nodes (sympy.combinatorics.prufer.Prufer attribute), 265
- NoneToken (class in sympy.codegen.ast), 402
- nonlinsolve() (in module sympy.solvers.solve), 1357
- nonnegative, 95
- nonnegative (sympy.assumptions.ask.AssumptionKeys attribute), 1082
- nonpositive, 95
- nonpositive (sympy.assumptions.ask.AssumptionKeys attribute), 1083
- NonSquareMatrixError (class in sympy.matrices.matrices), 754
- nonzero, 95
- nonzero (sympy.assumptions.ask.AssumptionKeys attribute), 1083
- Nor (class in sympy.logic.boolalg), 723
- norm (sympy.physics.quantum.state.Wavefunction attribute), 1687
- norm() (sympy.matrices.matrices.MatrixBase method), 748
- norm() (sympy.polys.polyclasses.DMP method), 921
- norm() (sympy.polys.polytools.Poly method), 848
- normal (sympy.assumptions.ask.AssumptionKeys attribute), 1084
- Normal() (in module sympy.stats), 1211
- normal_closure() (sympy.combinatorics.perm_groups.PermutationGroup method), 251
- normal_lines() (sympy.geometry.ellipse.Ellipse method), 619
- normal_vector (sympy.geometry.plane.Plane attribute), 653

`normalize()` (sympy.physics.quantum.state.Wavefunction method), 1687

`normalize()` (sympy.physics.vector.vector.Vector method), 1559

`normalize()` (sympy.vector.vector.Vector method), 1794

`normalize_theta_set()` (in module sympy.sets.fancysets), 1154

`normalized()` (in module sympy.physics.quantum.gate), 1693

`normalized()` (sympy.matrices.matrices.MatrixBase method), 749

`NormalPSpace` (class in sympy.stats.crv_types), 1229

`Not` (class in sympy.logic.boolalg), 722

`NotAlgebraic` (class in sympy.polys.polyerrors), 999

`NotInvertible` (class in sympy.polys.polyerrors), 999

`NotReversible` (class in sympy.polys.polyerrors), 999

`npartitions()` (in module sympy.ntheory.partitions_), 325

`nqubits` (sympy.physics.quantum.gate.CGate attribute), 1690

`nqubits` (sympy.physics.quantum.gate.Gate attribute), 1689

`nroots()` (in module sympy.polys.polytools), 819

`nroots()` (sympy.polys.polytools.Poly method), 848

`nseries()` (sympy.core.expr.Expr method), 129

`nsimplify()` (in module sympy.simplify.simplify), 1159

`nsimplify()` (sympy.core.expr.Expr method), 130

`nsolve()` (in module sympy.solvers.solvers), 1306

`nth()` (sympy.polys.polyclasses.DMP method), 921

`nth()` (sympy.polys.polytools.Poly method), 848

`nth_power_roots_poly()` (in module sympy.polys.polytools), 819

`nth_power_roots_poly()` (sympy.polys.polytools.Poly method), 849

`nthroot()` (in module sympy.simplify.simplify), 1158

`nthroot_mod()` (in module sympy.ntheory.residue_ntheory), 328

`nu` (sympy.functions.special.hyper.meijerg attribute), 534

`Num` (class in sympy.core.numbers), 138

`numbered_symbols()` (in module sympy.utilities.iterables), 1428

`NumberSymbol` (class in sympy.core.numbers), 143

`numer()` (sympy.polys.domains.AlgebraicField method), 914

`numer()` (sympy.polys.domains.domain.Domain method), 907

`numer()` (sympy.polys.domains.ExpressionDomain method), 917

`numer()` (sympy.polys.domains.FractionField method), 916

`numer()` (sympy.polys.domains.ring.Ring method), 910

`numer()` (sympy.polys.polyclasses.DMF method), 924

`NumPyPrinter` (class in sympy.printing.pycode), 1049

O

`Object` (class in sympy.categories), 1736

`objects` (sympy.categories.Category attribute), 1741

`objects` (sympy.categories.Diagram attribute), 1744

`octave_code()` (in module sympy.printing.octave), 1039

`OctaveCodeGen` (class in sympy.utilities.codegen), 1403

`OctaveCodePrinter` (class in sympy.printing.octave), 1039

`odd`, 95

`odd` (sympy.assumptions.ask.AssumptionKeys attribute), 1084

`ode_1st_exact()` (in module sympy.solvers.ode), 1242

`ode_1st_homogeneous_coeff_best()` (in module sympy.solvers.ode), 1243

`ode_1st_homogeneous_coeff_subs_dep_div_indep()` (in module sympy.solvers.ode), 1244

`ode_1st_homogeneous_coeff_subs_indep_div_dep()` (in module sympy.solvers.ode), 1245

`ode_1st_linear()` (in module sympy.solvers.ode), 1246

`ode_1st_power_series()` (in module sympy.solvers.ode), 1259

`ode_2nd_power_series_ordinary()` (in module sympy.solvers.ode), 1259

`ode_2nd_power_series_regular()` (in module sympy.solvers.ode), 1260

`ode_almost_linear()` (in module sympy.solvers.ode), 1255

ode_Bernoulli() (in module sympy.solvers.ode), 1247

ode_lie_group() (in module sympy.solvers.ode), 1258

ode_linear_coefficients() (in module sympy.solvers.ode), 1256

ode_Liouville() (in module sympy.solvers.ode), 1249

ode_nth_linear_constant_coeff_homogeneous() (in module sympy.solvers.ode), 1250

ode_nth_linear_constant_coeff_undetermined_coefficients() (in module sympy.solvers.ode), 1252

ode_nth_linear_constant_coeff_variation_of_parameters() (in module sympy.solvers.ode), 1253

ode_order() (in module sympy.solvers.deutils), 1310

ode_Riccati_special_minus2() (in module sympy.solvers.ode), 1250

ode_separable() (in module sympy.solvers.ode), 1254

ode_separable_reduced() (in module sympy.solvers.ode), 1257

ode_sol_simplicity() (in module sympy.solvers.ode), 1241

odesimp() (in module sympy.solvers.ode), 1238

of_type() (sympy.polys.domains.domain.Domain method), 907

offset (sympy.tensor.indexed.IndexedBase attribute), 1371

old_frac_field() (sympy.polys.domains.domain.Domain method), 907

old_poly_ring() (sympy.polys.domains.domain.Domain method), 907

One (class in sympy.core.numbers), 145

one (sympy.polys.polytools.Poly attribute), 849

OneQubitGate (class in sympy.physics.quantum.gate), 1690

ones() (in module sympy.matrices.dense), 755

open() (sympy.sets.sets.Interval class method), 1140

OperationNotSupported (class in sympy.polys.polyerrors), 999

Operator (class in sympy.physics.quantum.operator), 1658

operators (sympy.physics.quantum.state.StateBase attribute), 1681

operators_to_state() (in module sympy.physics.quantum.operatorset), 1663

opt_cse() (in module sympy.simplify.cse_main), 1175

Optimization (class in sympy.codegen.rewriting), 390

optimize() (in module sympy.codegen.rewriting), 391

OptionError (class in sympy.polys.polyerrors), 1000

Or (class in sympy.logic.boolalg), 721

OracleGate (class in sympy.physics.quantum.grover), 1694

orbit_transversal() (sympy.combinatorics.perm_groups.PermutationGroup method), 251

orbits() (sympy.combinatorics.perm_groups.PermutationGroup method), 252

Order (class in sympy.series.order), 1111

order (sympy.functions.special.bessel.BesselBase attribute), 512

order() (sympy.combinatorics.perm_groups.PermutationGroup method), 253

order() (sympy.combinatorics.permutations.Permutation method), 224

ordered_partitions() (in module sympy.utilities.iterables), 1428

orient() (sympy.physics.vector.frame.ReferenceFrame method), 1553

orient_new() (sympy.vector.coordsysrect.CoordSys3D method), 1787

orient_new_axis() (sympy.vector.coordsysrect.CoordSys3D method), 1788

orient_new_body() (sympy.vector.coordsysrect.CoordSys3D method), 1789

orient_new_quaternion() (sympy.vector.coordsysrect.CoordSys3D method), 1790

orient_new_space() (sympy.vector.coordsysrect.CoordSys3D method), 1791

Orienter (class in sympy.vector.orienters), 1798

orientnew() (sympy.physics.vector.frame.ReferenceFrame method), 1554

origin (sympy.geometry.point.Point attribute), 572

orthocenter (sympy.geometry.polygon.Triangle attribute), 649

orthogonal (sympy.assumptions.ask.AssumptionKeys attribute), 1084

orthogonal_direction (sympy.geometry.point.Point attribute), 572

`outer()` (in module `sympy.physics.vector.functions`), 1577

`outer()` (`sympy.physics.vector.vector.Vector` method), 1560

`outer()` (`sympy.vector.vector.Vector` method), 1794

`OuterProduct` (class in `sympy.physics.quantum.operator`), 1660

`OutputArgument` (class in `sympy.utilities.codegen`), 1399

P

`P()` (in module `sympy.stats`), 1221

`p1` (`sympy.geometry.line.LinearEntity` attribute), 585

`p1` (`sympy.geometry.plane.Plane` attribute), 654

`p2` (`sympy.geometry.line.LinearEntity` attribute), 585

`padded_key()` (in module `sympy.crypto.crypto`), 336

`parallel_line()` (`sympy.geometry.line.LinearEntity` method), 585

`parallel_plane()` (`sympy.geometry.plane.Plane` method), 654

`parallel_poly_from_expr()` (in module `sympy.polys.polytools`), 803

`parameter` (`sympy.geometry.curve.Curve` attribute), 610

`parameter_value()` (`sympy.geometry.entity.GeometryEntity` method), 562

`parameter_value()` (`sympy.geometry.plane.Plane` method), 654

`Parametric2DLineSeries` (class in `sympy.plotting.plot`), 1071

`Parametric3DLineSeries` (class in `sympy.plotting.plot`), 1072

`ParametricSurfaceSeries` (class in `sympy.plotting.plot`), 1072

`parametrize_ternary_quadratic()` (in module `sympy.solvers.diophantine`), 1335

`parens()` (`sympy.printing.pretty.stringpict.stringPict` method), 1056

`Pareto()` (in module `sympy.stats`), 1212

`parity()` (`sympy.combinatorics.permutations.Permutation` method), 225

`parse_expr()` (in module `sympy.parsing.sympy_parser`), 1455

`parse_latex()` (in module `sympy.parsing.latex`), 1459

`parse_maxima()` (in module `sympy.parsing.maxima`), 1456

`partial_velocity()` (in module `sympy.physics.vector.functions`), 1570

`partial_velocity()` (`sympy.physics.vector.frame.ReferenceFrame` method), 1555

`partial_velocity()` (`sympy.physics.vector.point.Point` method), 1566

`Particle` (class in `sympy.physics.mechanics.particle`), 1621

`Partition` (class in `sympy.combinatorics.partitions`), 202

`partition` (`sympy.combinatorics.partitions.Partition` attribute), 203

`partition()` (in module `sympy.solvers.diophantine`), 1330

`partitions()` (in module `sympy.utilities.iterables`), 1429

`pat_matrix()` (in module `sympy.physics.matrices`), 1473

`Patch` (class in `sympy.diffgeom`), 1754

`pde_1st_linear_constant_coeff()` (in module `sympy.solvers.pde`), 1292

`pde_1st_linear_constant_coeff_homogeneous()` (in module `sympy.solvers.pde`), 1291

`pde_1st_linear_variable_coeff()` (in module `sympy.solvers.pde`), 1294

`pde_separate()` (in module `sympy.solvers.pde`), 1287

`pde_separate_add()` (in module `sympy.solvers.pde`), 1288

`pde_separate_mul()` (in module `sympy.solvers.pde`), 1288

`pdiv()` (in module `sympy.polys.polytools`), 804

`pdiv()` (`sympy.polys.polyclasses.DMP` method), 921

`pdiv()` (`sympy.polys.polytools.Poly` method), 849

`pdsolve()` (in module `sympy.solvers.pde`), 1288

`per()` (`sympy.polys.polyclasses.DMF` method), 924

`per()` (`sympy.polys.polyclasses.DMP` method), 921

`per()` (`sympy.polys.polytools.Poly` method), 850

`perfect_power()` (in module `sympy.ntheory.factor_`), 308

`periapsis` (`sympy.geometry.ellipse.Ellipse` attribute), 620

`perimeter` (`sympy.geometry.polygon.Polygon` attribute), 632

`period_find()` (in module `sympy.physics.quantum.shor`), 1701

periodic_argument (class in PhaseGate (class in
 sympy.functions.elementary.complexes), sympy.physics.quantum.gate), 1691
 431 Pi (class in sympy.core.numbers), 149
 perm2tensor() (sympy.tensor.tensor.TensMul PIABBra (class in
 method), 1389 sympy.physics.quantum.piab), 1702
 permeability (sympy.physics.optics.medium.MediumPIABHamiltonian (class in
 attribute), 1714 sympy.physics.quantum.piab), 1701
 permittivity (sympy.physics.optics.medium.MediumPIABKet (class in
 attribute), 1714 sympy.physics.quantum.piab), 1701
 Permutation (class in Piecewise (class in
 sympy.combinatorics.permutations), sympy.functions.elementary.piecewise),
 207 450
 PermutationGroup (class in piecewise_fold() (in module
 sympy.combinatorics.perm_groups), sympy.functions.elementary.piecewise),
 232 451
 PermutationOperator (class in pinv() (sympy.matrices.matrices.MatrixBase
 sympy.physics.secondquant), 1496 method), 749
 permute() (sympy.polys.polyclasses.DMP pinv_solve() (sympy.matrices.matrices.MatrixBase
 method), 922 method), 750
 permute_signs() (in module Plane (class in sympy.geometry.plane), 650
 sympy.utilities.iterables), 1430 Plot (class in sympy.plotting.plot), 1059
 permutedims() (in module plot() (in module sympy.plotting.plot), 1061
 sympy.tensor.array), 1364 plot3d() (in module sympy.plotting.plot), 1065
 perpendicular_bisector() plot3d_parametric_line() (in module
 (sympy.geometry.line.Segment sympy.plotting.plot), 1067
 method), 597 plot3d_parametric_surface() (in module
 perpendicular_line() sympy.plotting.plot), 1068
 (sympy.geometry.line.LinearEntity plot_gate() (sympy.physics.quantum.gate.CGate
 method), 586 method), 1690
 perpendicular_line() plot_implicit() (in module
 (sympy.geometry.line.LinearEntity2D sympy.plotting.plot_implicit), 1069
 method), 598 plot_interval() (sympy.geometry.curve.Curve
 method), 610
 perpendicular_line() plot_interval() (sympy.geometry.ellipse.Ellipse
 (sympy.geometry.plane.Plane method), 620
 method), 655 plot_interval() (sympy.geometry.line.Line
 method), 591
 perpendicular_plane() plot_interval() (sympy.geometry.line.Ray
 (sympy.geometry.plane.Plane method), 593
 method), 655 plot_interval() (sympy.geometry.line.Segment
 method), 597
 perpendicular_segment() plot_interval() (sympy.geometry.polygon.Polygon
 (sympy.geometry.line.LinearEntity method), 632
 method), 586 plot_parametric() (in module
 pexquo() (in module sympy.polys.polytools), sympy.plotting.plot), 1063
 805 pexquo() (sympy.polys.polyclasses.DMP Point (class in sympy.diffgeom), 1756
 method), 922 Point (class in sympy.geometry.point), 567
 pexquo() (sympy.polys.polytools.Poly Point (class in sympy.physics.vector.point),
 method), 850 1564
 pgroup (sympy.combinatorics.polyhedron.Polyhedron point (sympy.core.function.Subs attribute),
 attribute), 262 186
 Phase (in module point (sympy.physics.mechanics.particle.Particle
 sympy.physics.quantum.gate), 1693 attribute), 1623
 phase (sympy.physics.optics.waves.TWave at-
 tribute), 1721

`point()` (sympy.diffgeom.CoordSystem method), 1756

`Point2D` (class in sympy.geometry.point), 573

`Point3D` (class in sympy.geometry.point), 575

`point_cflexure()` (sympy.physics.continuum_mechanics.beam.Beam method), 1728

`point_to_coords()` (sympy.diffgeom.CoordSystem method), 1756

`Pointer` (class in sympy.codegen.ast), 402

`points` (sympy.geometry.line.LinearEntity attribute), 587

`pointwise_stabilizer()` (sympy.combinatorics.perm_groups.PermGroup method), 254

`Poisson()` (in module sympy.stats), 1193

`polar` (sympy.sets.fancysets.ComplexRegion attribute), 1153

`polar_lift` (class in sympy.functions.elementary.complexes), 431

`PoleError` (class in sympy.core.function), 192

`PolificationFailed` (class in sympy.polys.polyerrors), 1000

`pollard_pm1()` (in module sympy.ntheory.factor_), 310

`pollard_rho()` (in module sympy.ntheory.factor_), 308

`Poly` (class in sympy.polys.polytools), 821

`poly()` (in module sympy.polys.polytools), 803

`poly_from_expr()` (in module sympy.polys.polytools), 803

`poly_ring()` (sympy.polys.domains.domain.Domain method), 907

`poly_unify()` (sympy.polys.polyclasses.DMF method), 924

`PolyElement` (class in sympy.polys.rings), 974

`polygamma` (class in sympy.functions.special.gamma_functions), 486

`Polygon` (class in sympy.geometry.polygon), 627

`Polyhedron` (class in sympy.combinatorics.polyhedron), 261

`polylog` (class in sympy.functions.special.zeta_functions), 526

`polynomial()` (sympy.series.formal.FormalPowerSeries method), 1124

`PolynomialError` (class in sympy.polys.polyerrors), 999

`PolynomialRing` (class in sympy.polys.domains), 912

`PolyRing` (class in sympy.polys.rings), 973

`polytope_integrate()` (in module sympy.integrals.intpoly), 702

`pos()` (sympy.polys.domains.domain.Domain method), 907

`posify()` (in module sympy.physics.vector.point.Point method), 1566

`POSform()` (in module sympy.logic.boolalg), 718

`posify()` (in module sympy.simplify.simplify), 1160

`position` (sympy.physics.quantum.cartesian.XBra attribute), 1656

`position` (sympy.physics.quantum.cartesian.XKet attribute), 1656

`position_wrt()` (sympy.vector.coordsysrect.CoordSys3D method), 1791

`position_x` (sympy.physics.quantum.cartesian.PositionState3D attribute), 1656

`position_y` (sympy.physics.quantum.cartesian.PositionState3D attribute), 1656

`position_z` (sympy.physics.quantum.cartesian.PositionState3D attribute), 1656

`PositionBra3D` (class in sympy.physics.quantum.cartesian), 1656

`PositionKet3D` (class in sympy.physics.quantum.cartesian), 1656

`PositionState3D` (class in sympy.physics.quantum.cartesian), 1656

`positive`, 95

`positive` (sympy.assumptions.ask.AssumptionKeys attribute), 1085

`positive_definite` (sympy.assumptions.ask.AssumptionKeys attribute), 1085

`positive_roots()` (sympy.liealgebras.type_a.TypeA method), 706

`positive_roots()` (sympy.liealgebras.type_b.TypeB method), 707

`positive_roots()` (sympy.liealgebras.type_c.TypeC method), 709

`positive_roots()` (sympy.liealgebras.type_d.TypeD method), 710

`positive_roots()` (sympy.liealgebras.type_e.TypeE method), 711

`positive_roots()` (sympy.liealgebras.type_f.TypeF method), 713

`positive_roots()` (sympy.liealgebras.type_g.TypeG method), 714

`PostDecrement` (class in sympy.codegen.cnodes), 413

`postfixes()` (in module sympy.utilities.iterables), 1430

PostIncrement (class in `sympy.codegen.cnodes`), 413
 postorder_traversal() (in module `sympy.utilities.iterables`), 1431
 potential_energy (sympy.physics.mechanics.particle.Particle attribute), 1623
 potential_energy (sympy.physics.mechanics.rigidbody.RigidBody attribute), 1626
 potential_energy() (in module `sympy.physics.mechanics.functions`), 1629
 Pow (class in `sympy.core.power`), 152
 Pow() (sympy.assumptions.handlers.calculus.AskFiniteHandler static method), 1097
 Pow() (sympy.assumptions.handlers.ntheory.AskPrimeHandler static method), 1097
 Pow() (sympy.assumptions.handlers.order.AskNegativeHandler static method), 1098
 Pow() (sympy.assumptions.handlers.sets.AskAntiHermitianHandler static method), 1098
 Pow() (sympy.assumptions.handlers.sets.AskHermitianHandler static method), 1099
 Pow() (sympy.assumptions.handlers.sets.AskImaginaryHandler static method), 1099
 Pow() (sympy.assumptions.handlers.sets.AskIntegerHandler static method), 1099
 Pow() (sympy.assumptions.handlers.sets.AskRationalHandler static method), 1099
 Pow() (sympy.assumptions.handlers.sets.AskRealHandler static method), 1099
 pow() (sympy.polys.domains.domain.Domain method), 907
 pow() (sympy.polys.polyclasses.ANP method), 925
 pow() (sympy.polys.polyclasses.DMF method), 924
 pow() (sympy.polys.polyclasses.DMP method), 922
 pow() (sympy.polys.polytools.Poly method), 850
 pow_xin() (in module `sympy.polys.ring_series`), 1019
 powdernest() (in module `sympy.simplify.powsimp`), 1171
 powerset() (sympy.sets.sets.Set method), 1136
 powsimp() (in module `sympy.simplify.powsimp`), 1170
 powsimp() (sympy.core.expr.Expr method), 130
 pprint_nodes() (in module `sympy.printing.tree`), 1050
 PQa() (in module `sympy.solvers.diophantine`), 1334
 pquo() (in module `sympy.polys.polytools`), 805
 pquo() (sympy.polys.polyclasses.DMP method), 922
 pquo() (sympy.polys.polytools.Poly method), 851
 PRECEDENCE (in module `sympy.printing.precedence`), 1054
 precedence() (in module `sympy.printing.precedence`), 1054
 PRECEDENCE_FUNCTIONS (in module `sympy.printing.precedence`), 1054
 PRECEDENCE_VALUES (in module `sympy.printing.precedence`), 1054
 PrecisionExhausted (class in `sympy.core.evalf`), 196
 PreDecrement (class in `sympy.codegen.cnodes`), 413
 Predicate (class in `sympy.assumptions.assume`), 1093
 preferred_index (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 557
 preferred_index (sympy.physics.secondquant.KroneckerDelta attribute), 1481
 Prefix (class in `sympy.physics.units.prefixes`), 142
 prefixes() (in module `sympy.utilities.iterables`), 1431
 PreIncrement (class in `sympy.codegen.cnodes`), 413
 prem() (in module `sympy.polys.polytools`), 804
 prem() (sympy.polys.polyclasses.DMP method), 922
 prem() (sympy.polys.polytools.Poly method), 851
 premises (sympy.categories.Diagram attribute), 1744
 presentation() (sympy.combinatorics.perm_groups.PermutationGroup method), 254
 pretty() (in module `sympy.printing.pretty.pretty`), 1023
 pretty_atom() (in module `sympy.printing.pretty.pretty_symbology`), 1055
 pretty_print() (in module `sympy.printing.pretty.pretty`), 1023
 pretty_symbol() (in module `sympy.printing.pretty.pretty_symbology`), 1055
 pretty_try_use_unicode() (in module `sympy.printing.pretty.pretty_symbology`), 1055
 pretty_use_unicode() (in module `sympy.printing.pretty.pretty_symbology`), 1055

`prettyForm` (class in `sympy.printing.pretty.stringpict`), 1057

`PrettyPrinter` (class in `sympy.printing.pretty.pretty`), 1023

`prev()` (`sympy.combinatorics.prufer.Prufer` method), 265

`prev_binary()` (`sympy.combinatorics.subsets.Subset` method), 270

`prev_gray()` (`sympy.combinatorics.subsets.Subset` method), 270

`prev_lex()` (`sympy.combinatorics.partitions.IntegerPartitions` method), 205

`prev_lexicographic()` (`sympy.combinatorics.subsets.Subset` method), 271

`preview()` (in module `sympy.printing.preview`), 1052

`preview_diagram()` (in module `sympy.categories.diagram_drawing`), 1753

`prevprime()` (in module `sympy.ntheory.generate`), 302

`prime`, 95

`prime` (`sympy.assumptions.ask.AssumptionKeys` attribute), 1086

`prime()` (in module `sympy.ntheory.generate`), 300

`prime_as_sum_of_two_squares()` (in module `sympy.solvers.diophantine`), 1338

`primefactors()` (in module `sympy.ntheory.factor_`), 314

`primenu` (class in `sympy.ntheory.factor_`), 321

`primeomega` (class in `sympy.ntheory.factor_`), 321

`primepi()` (in module `sympy.ntheory.generate`), 301

`primerange()` (in module `sympy.ntheory.generate`), 302

`primerange()` (`sympy.ntheory.generate.Sieve` method), 299

`primitive()` (in module `sympy.polys.polytools`), 814

`primitive()` (`sympy.core.add.Add` method), 160

`primitive()` (`sympy.core.expr.Expr` method), 130

`primitive()` (`sympy.polys.polyclasses.DMP` method), 922

`primitive()` (`sympy.polys.polytools.Poly` method), 851

`primitive()` (`sympy.polys.rings.PolyElement` method), 978

`primitive_element()` (in module `sympy.polys.numberfields`), 867

`primitive_root()` (in module `sympy.ntheory.residue_ntheory`), 327

`primorial()` (in module `sympy.ntheory.generate`), 304

`principal_branch` (class in `sympy.functions.elementary.complexes`), 432

`Print` (class in `sympy.codegen.ast`), 403

`printPartitions()` (in module `sympy.printing.ccode`), 1027

`print_dim_base()` (`sympy.physics.units.dimensions.Dimension` method), 1511

`print_fcode()` (in module `sympy.printing.fcode`), 1032

`print_gtk()` (in module `sympy.printing.gtk`), 1044

`print_latex()` (in module `sympy.printing.latex`), 1047

`print_mathml()` (in module `sympy.printing.mathml`), 1048

`print_node()` (in module `sympy.printing.tree`), 1051

`print_nonzero()` (`sympy.matrices.matrices.MatrixBase` method), 751

`print_rcode()` (in module `sympy.printing.rcode`), 1030

`print_tree()` (in module `sympy.printing.tree`), 1051

`print_unit_base()` (`sympy.physics.units.unitsystem.UnitSystem` method), 1512

`Printer` (class in `sympy.printing.printer`), 1022

`printmethod` (`sympy.printing.ccode.C89CodePrinter` attribute), 1024

`printmethod` (`sympy.printing.ccode.C99CodePrinter` attribute), 1024

`printmethod` (`sympy.printing.codeprinter.CodePrinter` attribute), 1054

`printmethod` (`sympy.printing.cxxcode.CXX11CodePrinter` attribute), 1027

`printmethod` (`sympy.printing.cxxcode.CXX98CodePrinter` attribute), 1027

`printmethod` (`sympy.printing.fcode.FCodePrinter` attribute), 1032

`printmethod` (`sympy.printing.jscode.JavascriptCodePrinter` attribute), 1035

`printmethod` (`sympy.printing.julia.JuliaCodePrinter` attribute), 1037

`printmethod` (`sympy.printing.lambdarepr.LambdaPrinter` attribute), 1045

`printmethod` (`sympy.printing.latex.LatexPrinter` attribute), 1045

[printmethod \(sympy.printing.mathematica.MCodePrinter attribute\), 1034](#)
[printmethod \(sympy.printing.mathml.MathMLPrinter attribute\), 1048](#)
[printmethod \(sympy.printing.mathml.MathMLPrinter attribute\), 1048](#)
[printmethod \(sympy.printing.octave.OctaveCodePrinter attribute\), 1039](#)
[printmethod \(sympy.printing.pretty.pretty.PrettyPrinter attribute\), 1023](#)
[printmethod \(sympy.printing.printer.Printer attribute\), 1022](#)
[printmethod \(sympy.printing.rcode.RCodePrinter attribute\), 1028](#)
[printmethod \(sympy.printing.repr.ReprPrinter attribute\), 1050](#)
[printmethod \(sympy.printing.rust.RustCodePrinter attribute\), 1042](#)
[printmethod \(sympy.printing.str.StrPrinter attribute\), 1050](#)
[printmethod \(sympy.printing.theanocode.TheanoPrinter attribute\), 1044](#)
[prob\(\) \(sympy.physics.quantum.state.Wavefunction method\), 1687](#)
[Probability \(class in sympy.stats\), 1221](#)
[prod\(\) \(in module sympy.core.mul\), 157](#)
[Product \(class in sympy.concrete.products\), 366](#)
[product\(\) \(in module sympy.concrete.products\), 372](#)
[product\(\) \(sympy.polys.agca.ideals.Ideal method\), 895](#)
[ProductDomain \(class in sympy.stats.rv\), 1228](#)
[ProductPSpace \(class in sympy.stats.rv\), 1228](#)
[ProductSet \(class in sympy.sets.sets\), 1143](#)
[Program \(class in sympy.codegen.fnodes\), 416](#)
[project\(\) \(sympy.geometry.point.Point static method\), 572](#)
[project\(\) \(sympy.matrices.matrices.MatrixBase method\), 751](#)
[projection\(\) \(sympy.geometry.line.LinearEntity method\), 587](#)
[projection\(\) \(sympy.geometry.plane.Plane method\), 656](#)
[projection\(\) \(sympy.vector.vector.Vector method\), 1794](#)
[projection_line\(\) \(sympy.geometry.plane.Plane method\), 656](#)
[Prufer \(class in sympy.combinatorics.prufer\), 264](#)
[prufer_rank\(\) \(sympy.combinatorics.prufer.Prufer method\), 265](#)
[PruferPrinter \(sympy.combinatorics.prufer.Prufer attribute\), 265](#)
[PrettyPrinter \(sympy.pretty.pretty.PrettyPrinter attribute\), 1153](#)
[PresentationPrinter \(sympy.physics.qho_1d\), 1475](#)
[PSpace \(in module sympy.physics.hydrogen\), 1470](#)
[PSpacePrinter \(class in sympy.stats.rv\), 1228](#)
[pSPACE\(\) \(in module sympy.stats.rv\), 1229](#)
[public\(\) \(in module sympy.utilities.decorator\), 1409](#)
[PurePoly \(class in sympy.polys.polytools\), 861](#)
[PxBra \(class in sympy.physics.quantum.cartesian\), 1656](#)
[PxKet \(class in sympy.physics.quantum.cartesian\), 1656](#)
[PxOp \(class in sympy.physics.quantum.cartesian\), 1656](#)
[pycode\(\) \(in module sympy.printing.pycode\), 1049](#)
[PyTestReporter \(class in sympy.utilities.runtests\), 1447](#)
[Python Enhancement Proposals PEP 335, 1854, 1855](#)
[PythonFiniteField \(class in sympy.polys.domains\), 917](#)
[PythonIntegerRing \(class in sympy.polys.domains\), 918](#)
[PythonRationalField \(class in sympy.polys.domains\), 918](#)

Q

[q \(sympy.physics.optics.gaussopt.BeamParameter attribute\), 1709](#)
[qapply\(\) \(in module sympy.physics.quantum.qapply\), 1665](#)
[QFT \(class in sympy.physics.quantum.qft\), 1696](#)
[QRdecomposition\(\) \(sympy.matrices.matrices.MatrixBase method\), 742](#)
[QRsolve\(\) \(sympy.matrices.matrices.MatrixBase method\), 742](#)
[quadratic_residues\(\) \(in module sympy.ntheory.residue_ntheory\), 328](#)
[QuadraticU\(\) \(in module sympy.stats\), 1212](#)
[Quantity \(class in sympy.physics.units.quantities\), 1513](#)
[QuaternionOrienter \(class in sympy.vector.orienters\), 1801](#)

- Qubit (class in `sympy.physics.quantum.qubit`), 1696
- `qubit_to_matrix()` (in module `sympy.physics.quantum.qubit`), 1698
- QubitBra (class in `sympy.physics.quantum.qubit`), 1697
- `quo()` (in module `sympy.polys.polytools`), 806
- `quo()` (`sympy.polys.domains.domain.Domain` method), 907
- `quo()` (`sympy.polys.domains.field.Field` method), 909
- `quo()` (`sympy.polys.domains.ring.Ring` method), 910
- `quo()` (`sympy.polys.polyclasses.DMF` method), 924
- `quo()` (`sympy.polys.polyclasses.DMP` method), 922
- `quo()` (`sympy.polys.polytools.Poly` method), 851
- `quo_ground()` (`sympy.polys.polyclasses.DMP` method), 922
- `quo_ground()` (`sympy.polys.polytools.Poly` method), 852
- QuotedString (class in `sympy.codegen.ast`), 403
- `quotient()` (`sympy.polys.agca.ideals.Ideal` method), 895
- `quotient_codomain()` (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 902
- `quotient_domain()` (`sympy.polys.agca.homomorphisms.ModuleHomomorphism` method), 903
- `quotient_hom()` (`sympy.polys.agca.modules.QuotientModule` method), 897
- `quotient_hom()` (`sympy.polys.agca.modules.SubQuotientModule` method), 898
- `quotient_module()` (`sympy.polys.agca.modules.FreeModule` method), 889
- `quotient_module()` (`sympy.polys.agca.modules.Module` method), 588
- `quotient_module()` (`sympy.polys.agca.modules.SubModule` method), 892
- `quotient_ring()` (`sympy.polys.domains.ring.Ring` method), 910
- QuotientModule (class in `sympy.polys.agca.modules`), 896
- R**
- `R_nl()` (in module `sympy.physics.hydrogen`), 1471
- `R_nl()` (in module `sympy.physics.sho`), 1476
- `racah()` (in module `sympy.physics.wigner`), 1500
- `rad_rationalize()` (in module `sympy.simplify.radsimp`), 1163
- `radical()` (`sympy.polys.agca.ideals.Ideal` method), 895
- `radius` (`sympy.geometry.ellipse.Circle` attribute), 626
- `radius` (`sympy.geometry.polygon.RegularPolygon` attribute), 639
- `radius` (`sympy.physics.optics.gaussopt.BeamParameter` attribute), 1709
- `radius_of_convergence` (`sympy.functions.special.hyper.hyper` attribute), 531
- `radsimp()` (in module `sympy.simplify.radsimp`), 1162
- `radsimp()` (`sympy.core.expr.Expr` method), 130
- `RaisedCosine()` (in module `sympy.stats`), 1213
- `raises()` (in module `sympy.utilities.pytest`), 1445
- `randMatrix()` (in module `sympy.matrices.dense`), 758
- `random()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 254
- `random()` (`sympy.combinatorics.permutations.Permutation` class method), 225
- `random_bitstring()` (`sympy.combinatorics.graycode` method), 277
- `random_circuit()` (in module `sympy.physics.quantum.gate`), 1693
- `random_module_homomorphism()` (in module `sympy.utilities.randtest`), 1446
- `random_integer_partition()` (in module `sympy.combinatorics.partitions`), 255
- `random_point()` (`sympy.geometry.ellipse.Ellipse` method), 620
- `random_point()` (`sympy.geometry.line.LinearEntity` method), 620
- `random_point()` (`sympy.geometry.plane.Plane` method), 657
- `random_poly()` (in module `sympy.polys.specialpolys`), 873
- `random_pr()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 254
- `random_stab()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 254
- `random_symbols()` (in module `sympy.stats.rv`), 1229
- RandomDomain (class in `sympy.stats.rv`), 1228
- RandomSymbol (class in `sympy.stats.rv`), 1228

randprime() (in module [1711](#)
 sympy.ntheory.generate), [303](#) RayTransferMatrix (class in
 Range (class in sympy.sets.fancysets), [1149](#) sympy.physics.optics.gaussopt),
 ranges (sympy.tensor.indexed.Indexed at- [1702](#)
 tribute), [1369](#) rcall() (sympy.core.basic.Basic method), [104](#)
 rank (sympy.combinatorics.graycode.GrayCode code() (in module sympy.printing.rcode),
 attribute), [276](#) [1028](#)
 rank (sympy.combinatorics.partitions.Partition RCodePrinter (class in sympy.printing.rcode),
 attribute), [203](#) [1028](#)
 rank (sympy.combinatorics.prufer.Prufer at- rcollect() (in module sympy.simplify.radsimp),
 tribute), [266](#) [1165](#)
 rank (sympy.tensor.indexed.Indexed at- re (class in sympy.functions.elementary.complexes),
 tribute), [1370](#) [427](#)
 rank() (sympy.combinatorics.permutations.Permutation method), [225](#) [1729](#)
 rank() (sympy.liealgebras.cartan_type.StandardCartan method), [716](#) [1087](#)
 rank_binary (sympy.combinatorics.subsets.Subset attribute), [271](#) [1087](#)
 rank_gray (sympy.combinatorics.subsets.Subset attribute), [271](#) [1087](#)
 rank_lexicographic (sympy.combinatorics.subsets.Subset attribute), [271](#) [458](#)
 rank_nonlex() (sympy.combinatorics.permutations.Permutation method), [226](#) [818](#)
 rank_trotterjohnson() real_roots() (in module
 (sympy.combinatorics.permutations.Permutation method), [226](#) [852](#)
 method), [226](#) RealField (class in sympy.polys.domains), [916](#)
 rat_clear_denoms() (sympy.polys.polytools.Poly RealNumber (in module
 method), [852](#) sympy.core.numbers), [144](#)
 rational, [95](#) reconstruct() (in module
 Rational (class in sympy.core.numbers), [141](#) sympy.solvers.diophantine), [1339](#)
 rational (sympy.assumptions.ask.AssumptionKeys currency_memo() (in module
 attribute), [1086](#) sympy.utilities.memoization), [1441](#)
 rational_algorithm() (in module red_groebner() (in module
 sympy.series.formal), [1126](#) sympy.polys.groebnertools), [997](#)
 rational_independent() (in module reduce() (sympy.polys.polytools.GroebnerBasis
 sympy.series.formal), [1126](#) method), [863](#)
 RationalField (class in sympy.polys.domains), [913](#) reduce() (sympy.series.sequences.SeqAdd
 static method), [1119](#)
 rationalize() (in module reduce() (sympy.series.sequences.SeqMul
 sympy.parsing.sympy_parser), [1458](#) static method), [1119](#)
 ratsimp() (in module sympy.simplify.ratsimp), [1169](#) reduce() (sympy.sets.sets.Complement static
 method), [1145](#)
 ratsimp() (sympy.core.expr.Expr method), [131](#) reduce_abs_inequalities() (in module
 sympy.solvers.inequalities), [1340](#)
 rawlines() (in module sympy.utilities.misc), [1442](#) reduce_abs_inequality() (in module
 sympy.solvers.inequalities), [1340](#)
 Ray (class in sympy.geometry.line), [591](#) reduce_element() (sympy.polys.agca.ideals.Ideal
 method), [895](#)
 Ray2D (class in sympy.geometry.line), [600](#) reduce_element() (sympy.polys.agca.modules.SubModule
 method), [892](#)
 Ray3D (class in sympy.geometry.line), [605](#) reduce_inequalities() (in module
 sympy.solvers.inequalities), [1341](#)
 Rayleigh() (in module sympy.stats), [1214](#)
 rayleigh2waist() (in module
 sympy.physics.optics.gaussopt),

`reduce_rational_inequalities()` (in module `sympy.solvers.inequalities`), 1340

`reduced()` (in module `sympy.polys.polytools`), 820

`reduced_totient` (class in `sympy.ntheory.factor_`), 317

`ReferenceFrame` (class in `sympy.physics.vector.frame`), 1552

`refine()` (in module `sympy.assumptions.refine`), 1094

`refine()` (`sympy.core.expr.Expr` method), 131

`refine_abs()` (in module `sympy.assumptions.refine`), 1094

`refine_atan2()` (in module `sympy.assumptions.refine`), 1095

`refine_Pow()` (in module `sympy.assumptions.refine`), 1094

`refine_Relational()` (in module `sympy.assumptions.refine`), 1094

`refine_root()` (in module `sympy.polys.polytools`), 818

`refine_root()` (`sympy.polys.polyclasses.DMP` method), 922

`refine_root()` (`sympy.polys.polytools.Poly` method), 853

`RefinementFailed` (class in `sympy.polys.polyerrors`), 999

`reflect()` (`sympy.geometry.ellipse.Circle` method), 626

`reflect()` (`sympy.geometry.ellipse.Ellipse` method), 621

`reflect()` (`sympy.geometry.entity.GeometryEntity` method), 562

`reflect()` (`sympy.geometry.polygon.RegularPolygon` method), 640

`refraction_angle()` (in module `sympy.physics.optics.utils`), 1715

`refractive_index` (`sympy.physics.optics.medium.Medium` attribute), 1715

`register_handler()` (in module `sympy.assumptions.ask`), 1091

`RegularPolygon` (class in `sympy.geometry.polygon`), 634

`Rel` (in module `sympy.core.relational`), 161

`rem()` (in module `sympy.polys.polytools`), 805

`rem()` (`sympy.polys.domains.domain.Domain` method), 907

`rem()` (`sympy.polys.domains.field.Field` method), 909

`rem()` (`sympy.polys.domains.ring.Ring` method), 910

`rem()` (`sympy.polys.polyclasses.DMP` method), 922

`rem()` (`sympy.polys.polytools.Poly` method), 853

`remove_handler()` (in module `sympy.assumptions.ask`), 1092

`remove_load()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1729

`removeO()` (`sympy.core.expr.Expr` method), 131

`render()` (`sympy.printing.pretty.stringpict.StringPict` method), 1056

`render_as_module()` (in module `sympy.codegen.futils`), 423

`render_as_module()` (in module `sympy.codegen.pyutils`), 422

`render_as_source_file()` (in module `sympy.codegen.cutils`), 423

`reorder()` (`sympy.polys.polytools.Poly` method), 853

`rep_expectation()` (in module `sympy.physics.quantum.represent`), 1668

`rep_innerproduct()` (in module `sympy.physics.quantum.represent`), 1667

`repeated_decimals()` (in module `sympy.parsing.sympy_parser`), 1458

`replace()` (in module `sympy.utilities.misc`), 1443

`replace()` (`sympy.core.basic.Basic` method), 104

`replace()` (`sympy.polys.polytools.Poly` method), 853

`ReplaceOptim` (class in `sympy.codegen.rewriting`), 390

`Reporter` (class in `sympy.utilities.runtests`), 1447

`represent()` (in module `sympy.physics.quantum.represent`), 1666

`reprify()` (`sympy.printing.repr.ReprPrinter` method), 1050

`ReprPrinter` (class in `sympy.printing.repr`), 1050

`reset()` (`sympy.combinatorics.polyhedron.Polyhedron` method), 262

`reshape()` (in module `sympy.codegen.fnodes`), 419

`reshape()` (in module `sympy.utilities.iterables`), 1431

`residue()` (in module `sympy.series.residues`), 1114

`restrict_codomain()` (`sympy.polys.agca.homomorphisms.Morphism` method), 903

[restrict_domain\(\)](#) (sympy.polys.agca.homomorphisms.homomorphisms.lagrange.LagrangeMethod method), 903
[Result](#) (class in sympy.utilities.codegen), 1399
[result_variables](#) (sympy.utilities.codegen.Routine attribute), 1399
[resultant\(\)](#) (in module sympy.polys.polytools), 808
[resultant\(\)](#) (sympy.polys.polyclasses.DMP method), 922
[resultant\(\)](#) (sympy.polys.polytools.Poly method), 854
[retract\(\)](#) (sympy.polys.polytools.Poly method), 854
[Return](#) (class in sympy.codegen.ast), 403
[reverse_order\(\)](#) (sympy.concrete.products.Product method), 370
[reverse_order\(\)](#) (sympy.concrete.summations.Sum method), 365
[reversed](#) (sympy.sets.fancysets.Range attribute), 1150
[ReversedGradedLexOrder](#) (class in sympy.polys.orderings), 869
[revert\(\)](#) (sympy.polys.domains.domain.Domain method), 907
[revert\(\)](#) (sympy.polys.domains.field.Field method), 909
[revert\(\)](#) (sympy.polys.domains.ring.Ring method), 910
[revert\(\)](#) (sympy.polys.polyclasses.DMP method), 922
[revert\(\)](#) (sympy.polys.polytools.Poly method), 854
[rewrite\(\)](#) (in module sympy.series.gruntz), 1108
[rewrite\(\)](#) (sympy.core.basic.Basic method), 106
[RGS](#) (sympy.combinatorics.partitions.Partition attribute), 202
[RGS_enum\(\)](#) (in module sympy.combinatorics.partitions), 206
[RGS_generalized\(\)](#) (in module sympy.combinatorics.partitions), 206
[RGS_rank\(\)](#) (in module sympy.combinatorics.partitions), 207
[RGS_unrank\(\)](#) (in module sympy.combinatorics.partitions), 207
[rhs\(\)](#) (sympy.physics.mechanics.kane.KanesMethod method), 1634
[rhs\(\)](#) (sympy.physics.mechanics.lagrange.LagrangeMethod method), 1637
[richardson\(\)](#) (in module sympy.series.acceleration), 1112
[riemann_cyclic\(\)](#) (in module sympy.tensor.tensor), 1390
[riemann_cyclic_replace\(\)](#) (in module sympy.tensor.tensor), 1390
[right](#) (sympy.sets.sets.Interval attribute), 1140
[right\(\)](#) (sympy.printing.pretty.stringpict.stringPict method), 1057
[right_open](#) (sympy.sets.sets.Interval attribute), 1140
[RigidBody](#) (class in sympy.physics.mechanics.rigidbody), 1623
[Ring](#) (class in sympy.polys.domains.ring), 909
[ring\(\)](#) (in module sympy.polys.rings), 971
[RisingFactorial](#) (class in sympy.functions.combinatorial.factorials), 472
[Rk](#) (in module sympy.physics.quantum.qft), 1696
[RkGate](#) (class in sympy.physics.quantum.qft), 1696
[RL](#) (sympy.matrices.sparse.SparseMatrix attribute), 768
[rmul\(\)](#) (sympy.combinatorics.permutations.Permutation static method), 226
[rmul_with_af\(\)](#) (sympy.combinatorics.permutations.Permutation class method), 227
[root](#) (in module sympy.printing.pretty.pretty_symbology), 1055
[root\(\)](#) (in module sympy.functions.elementary.miscellaneous), 454
[root\(\)](#) (sympy.polys.polytools.Poly method), 855
[root\(\)](#) (sympy.printing.pretty.stringpict.stringPict method), 1057
[root_space\(\)](#) (sympy.liealgebras.root_system.RootSystem method), 704
[RootOf](#) (class in sympy.polys.rootoftools), 870
[rootof\(\)](#) (in module sympy.polys.rootoftools), 869
[roots\(\)](#) (in module sympy.polys.polyroots), 871
[roots\(\)](#) (sympy.liealgebras.type_a.TypeA method), 706
[roots\(\)](#) (sympy.liealgebras.type_b.TypeB method), 707
[roots\(\)](#) (sympy.liealgebras.type_c.TypeC method), 709

`roots()` (sympy.liealgebras.type_d.TypeD method), 710

`roots()` (sympy.liealgebras.type_e.TypeE method), 712

`roots()` (sympy.liealgebras.type_f.TypeF method), 713

`roots()` (sympy.liealgebras.type_g.TypeG method), 714

`RootSum` (class in sympy.polys.rootoftools), 871

`RootSystem` (class in sympy.liealgebras.root_system), 703

`Ropen()` (sympy.sets.sets.Interval class method), 1139

`rot_axis1()` (in module sympy.matrices.dense), 760

`rot_axis2()` (in module sympy.matrices.dense), 761

`rot_axis3()` (in module sympy.matrices.dense), 761

`rotate()` (sympy.combinatorics.polyhedron.Polyhedron method), 262

`rotate()` (sympy.geometry.curve.Curve method), 611

`rotate()` (sympy.geometry.ellipse.Ellipse method), 622

`rotate()` (sympy.geometry.entity.GeometryEntity method), 563

`rotate()` (sympy.geometry.point.Point2D method), 574

`rotate()` (sympy.geometry.polygon.RegularPolygon method), 640

`rotate_left()` (in module sympy.utilities.iterables), 1432

`rotate_right()` (in module sympy.utilities.iterables), 1432

`Rotation` (class in sympy.physics.quantum.spin), 1670

`rotation` (sympy.geometry.polygon.RegularPolygon attribute), 640

`rotation_matrix()` (sympy.vector.coordsysrect.CoordSys3D method), 1792

`rotation_matrix()` (sympy.vector.orienters.AxisOriented method), 1799

`rotation_matrix()` (sympy.vector.orienters.Oriented method), 1798

`round()` (sympy.core.expr.Expr method), 131

`RoundFunction` (class in sympy.functions.elementary.integers), 448

`Routine` (class in sympy.utilities.codegen), 1398

`routine()` (sympy.utilities.codegen.CodeGen method), 1400

`routine()` (sympy.utilities.codegen.JuliaCodeGen method), 1403

`routine()` (sympy.utilities.codegen.OctaveCodeGen method), 1404

`routine()` (sympy.utilities.codegen.RustCodeGen method), 1405

`row_del()` (sympy.matrices.dense.MutableDenseMatrix method), 765

`row_del()` (sympy.matrices.sparse.MutableSparseMatrix method), 774

`row_join()` (sympy.matrices.sparse.MutableSparseMatrix method), 774

`row_list()` (sympy.matrices.sparse.SparseMatrix method), 770

`row_op()` (sympy.matrices.dense.MutableDenseMatrix method), 765

`row_op()` (sympy.matrices.sparse.MutableSparseMatrix method), 775

`row_structure_symbolic_cholesky()` (sympy.matrices.sparse.SparseMatrix method), 770

`row_swap()` (sympy.matrices.dense.MutableDenseMatrix method), 765

`row_swap()` (sympy.matrices.sparse.MutableSparseMatrix method), 775

`rs_asin()` (in module sympy.polys.ring_series), 1009

`rs_atan()` (in module sympy.polys.ring_series), 1009

`rs_atanh()` (in module sympy.polys.ring_series), 1011

`rs_compose_add()` (in module sympy.polys.ring_series), 1017

`rs_cos()` (in module sympy.polys.ring_series), 1010

`rs_cos_sin()` (in module sympy.polys.ring_series), 1011

`rs_cosh()` (in module sympy.polys.ring_series), 1012

`rs_cosh_sin()` (in module sympy.polys.ring_series), 1010

`rs_diff()` (in module sympy.polys.ring_series), 1016

`rs_exp()` (in module sympy.polys.ring_series), 1009

`rs_fun()` (in module sympy.polys.ring_series), 1018

`rs_hadamard_exp()` (in module sympy.polys.ring_series), 1012

`rs_integrate()` (in module sympy.polys.ring_series), 1016

- [rs_is_puiseux\(\)](#) (in module [sympy.polys.ring_series](#)), 1017
[rs_LambertW\(\)](#) (in module [sympy.polys.ring_series](#)), 1008
[rs_log\(\)](#) (in module [sympy.polys.ring_series](#)), 1008
[rs_mul\(\)](#) (in module [sympy.polys.ring_series](#)), 1013
[rs_newton\(\)](#) (in module [sympy.polys.ring_series](#)), 1016
[rs_nth_root\(\)](#) (in module [sympy.polys.ring_series](#)), 1014
[rs_pow\(\)](#) (in module [sympy.polys.ring_series](#)), 1013
[rs_puiseux\(\)](#) (in module [sympy.polys.ring_series](#)), 1017
[rs_puiseux2\(\)](#) (in module [sympy.polys.ring_series](#)), 1018
[rs_series_from_list\(\)](#) (in module [sympy.polys.ring_series](#)), 1018
[rs_series_inversion\(\)](#) (in module [sympy.polys.ring_series](#)), 1013
[rs_series_reversion\(\)](#) (in module [sympy.polys.ring_series](#)), 1014
[rs_sin\(\)](#) (in module [sympy.polys.ring_series](#)), 1010
[rs_sinh\(\)](#) (in module [sympy.polys.ring_series](#)), 1011
[rs_square\(\)](#) (in module [sympy.polys.ring_series](#)), 1013
[rs_subs\(\)](#) (in module [sympy.polys.ring_series](#)), 1015
[rs_swap\(\)](#) (in module [sympy.stats.rv](#)), 1229
[rs_tan\(\)](#) (in module [sympy.polys.ring_series](#)), 1009
[rs_tanh\(\)](#) (in module [sympy.polys.ring_series](#)), 1012
[rs_trunc\(\)](#) (in module [sympy.polys.ring_series](#)), 1015
[rsa_private_key\(\)](#) (in module [sympy.crypto.crypto](#)), 349
[rsa_public_key\(\)](#) (in module [sympy.crypto.crypto](#)), 349
[rsolve\(\)](#) (in module [sympy.solvers.recurr](#)), 1310
[rsolve_hyper\(\)](#) (in module [sympy.solvers.recurr](#)), 1312
[rsolve_hypergeometric\(\)](#) (in module [sympy.series.formal](#)), 1128
[rsolve_poly\(\)](#) (in module [sympy.solvers.recurr](#)), 1311
[rsolve_ratio\(\)](#) (in module [sympy.solvers.recurr](#)), 1311
[run\(\)](#) ([sympy.utilities.runtests.SymPyDocTestRunner](#) method), 1447
[run_all_tests\(\)](#) (in module [sympy.utilities.runtests](#)), 1449
[run_in_subprocess_with_hash_randomization\(\)](#) (in module [sympy.utilities.runtests](#)), 1449
[runs\(\)](#) (in module [sympy.utilities.iterables](#)), 1433
[runs\(\)](#) ([sympy.combinatorics.permutations.Permutation](#) method), 227
[rust_code\(\)](#) (in module [sympy.printing.rust](#)), 1042
[RustCodeGen](#) (class in [sympy.utilities.codegen](#)), 1404
[RustCodePrinter](#) (class in [sympy.printing.rust](#)), 1042
- ## S
- [S](#) (in module [sympy.physics.quantum.gate](#)), 1693
[sample\(\)](#) (in module [sympy.stats](#)), 1227
[sample_iter\(\)](#) (in module [sympy.stats](#)), 1227
[satisfiable\(\)](#) (in module [sympy.logic.inference](#)), 728
[saturate\(\)](#) ([sympy.polys.agca.ideals.Ideal](#) method), 895
[scalar_map\(\)](#) ([sympy.vector.coordsysrect.CoordSys3D](#) method), 1792
[scalar_multiply\(\)](#) ([sympy.matrices.sparse.SparseMatrix](#) method), 771
[scalar_potential\(\)](#) (in module [sympy.physics.vector.fieldfunctions](#)), 1580
[scalar_potential\(\)](#) (in module [sympy.vector](#)), 1805
[scalar_potential_difference\(\)](#) (in module [sympy.physics.vector.fieldfunctions](#)), 1581
[scalar_potential_difference\(\)](#) (in module [sympy.vector](#)), 1806
[scale\(\)](#) ([sympy.geometry.curve.Curve](#) method), 611
[scale\(\)](#) ([sympy.geometry.ellipse.Circle](#) method), 626
[scale\(\)](#) ([sympy.geometry.ellipse.Ellipse](#) method), 622
[scale\(\)](#) ([sympy.geometry.entity.GeometryEntity](#) method), 563
[scale\(\)](#) ([sympy.geometry.point.Point2D](#) method), 574
[scale\(\)](#) ([sympy.geometry.point.Point3D](#) method), 578

`scale()` (`sympy.geometry.polygon.RegularPolygon` method), 640

`scale()` (`sympy.series.fourier.FourierSeries` method), 1120

`scale_factor` (`sympy.physics.units.quantities.Quantity` attribute), 1513

`scalex()` (`sympy.series.fourier.FourierSeries` method), 1120

`schreier_sims()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 254

`schreier_sims_incremental()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 255

`schreier_sims_random()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 256

`schreier_vector()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 257

`Scope` (class in `sympy.codegen.ast`), 403

`sdm_add()` (in module `sympy.polys.distributedmodules`), 981

`sdm_deg()` (in module `sympy.polys.distributedmodules`), 982

`sdm_ecart()` (in module `sympy.polys.distributedmodules`), 998

`sdm_from_dict()` (in module `sympy.polys.distributedmodules`), 981

`sdm_from_vector()` (in module `sympy.polys.distributedmodules`), 983

`sdm_groebner()` (in module `sympy.polys.distributedmodules`), 999

`sdm_LC()` (in module `sympy.polys.distributedmodules`), 981

`sdm_LM()` (in module `sympy.polys.distributedmodules`), 981

`sdm_LT()` (in module `sympy.polys.distributedmodules`), 982

`sdm_monomial_deg()` (in module `sympy.polys.distributedmodules`), 980

`sdm_monomial_divides()` (in module `sympy.polys.distributedmodules`), 980

`sdm_monomial_mul()` (in module `sympy.polys.distributedmodules`), 979

`sdm_mul_term()` (in module `sympy.polys.distributedmodules`), 982

`sdm_poly()` (in module `sympy.polys.distributedmodules`), 997

`sdm_to_dict()` (in module `sympy.polys.distributedmodules`), 981

`sdm_to_vector()` (in module `sympy.polys.distributedmodules`), 983

`sdm_to_perm()` (in module `sympy.polys.distributedmodules`), 982

`search()` (`sympy.ntheory.generate.Sieve` method), 300

`search_function` (`sympy.physics.quantum.grover.OracleGate` attribute), 1694

`sec` (class in `sympy.functions.elementary.trigonometric`), 435

`sech` (class in `sympy.functions.elementary.hyperbolic`), 444

`second_moment` (`sympy.physics.continuum_mechanics.beam.Beam` attribute), 1730

`second_moment_of_area()` (`sympy.geometry.ellipse.Ellipse` method), 622

`second_moment_of_area()` (`sympy.geometry.polygon.Polygon` method), 632

`Segment` (class in `sympy.geometry.line`), 594

`Segment2D` (class in `sympy.geometry.line`), 602

`Segment3D` (class in `sympy.geometry.line`), 607

`select()` (`sympy.simplify.epathtools.EPath` method), 1178

`selections` (`sympy.combinatorics.graycode.GrayCode` attribute), 276

`semilatus_rectum` (`sympy.geometry.ellipse.Ellipse` attribute), 623

`separate()` (`sympy.core.expr.Expr` method), 131

`separate()` (`sympy.physics.vector.vector.Vector` method), 1560

`separate()` (`sympy.vector.vector.Vector` method), 1795

`separatevars()` (in module `sympy.simplify.simplify`), 1157

SeqAdd (class in sympy.series.sequences), 1118

SeqBase (class in sympy.series.sequences), 1115

SeqFormula (class in sympy.series.sequences), 1116

SeqMul (class in sympy.series.sequences), 1119

SeqPer (class in sympy.series.sequences), 1117

sequence() (in module sympy.series.sequences), 1114

series() (in module sympy.series.series), 1110

series() (sympy.core.expr.Expr method), 131

series() (sympy.holonomic.holonomic.HolonomicFunction method), 661

series() (sympy.liealgebras.cartan_type.StandardCartan method), 716

SeriesApprox (class in sympy.codegen.approximations), 391

Set (class in sympy.sets.sets), 1132

set_acc() (sympy.physics.vector.point.Point method), 1567

set_ang_acc() (sympy.physics.vector.frame.ReferenceFrame method), 1555

set_ang_vel() (sympy.physics.vector.frame.ReferenceFrame method), 1556

set_comm() (sympy.tensor.tensor._TensorManager method), 1376

set_comms() (sympy.tensor.tensor._TensorManager method), 1376

set_domain() (sympy.polys.polytools.Poly method), 855

set_global_settings() (sympy.printing.printer.Printer class method), 1023

set_modulus() (sympy.polys.polytools.Poly method), 855

set_pos() (sympy.physics.vector.point.Point method), 1567

set_vel() (sympy.physics.vector.point.Point method), 1567

seterr() (in module sympy.core.numbers), 144

sets (sympy.sets.fancysets.ComplexRegion attribute), 1153

shanks() (in module sympy.series.acceleration), 1113

shape (sympy.tensor.indexed.Indexed attribute), 1370

shape (sympy.tensor.indexed.IndexedBase attribute), 1372

shape() (in module sympy.codegen.fnodes), 420

ShapeError (class in sympy.matrices.matrices), 754

shear_force() (sympy.physics.continuum_mechanics.beam.Beam method), 1730

Shi (class in sympy.functions.special.error_functions), 509

shift() (sympy.polys.polyclasses.DMP method), 922

shift() (sympy.polys.polytools.Poly method), 856

shift() (sympy.series.fourier.FourierSeries method), 1120

shiftx() (sympy.series.fourier.FourierSeries method), 1121

ShorFunction (in module sympy.physics.quantum.shor), 1701

Si (class in sympy.functions.special.error_functions), 507

sides (sympy.geometry.polygon.Polygon attribute), 633

Sieve (class in sympy.ntheory.generate), 298

sift() (in module sympy.utilities.iterables), 1433

sigma_approximation() (sympy.series.fourier.FourierSeries method), 1121

sign (class in sympy.functions.elementary.complexes), 428

sign() (in module sympy.series.gruntz), 1108

signature() (sympy.combinatorics.permutations.Permutation method), 227

signed_permutations() (in module sympy.utilities.iterables), 1434

SignedIntType (class in sympy.codegen.ast), 403

simple_root() (sympy.liealgebras.type_a.TypeA method), 706

simple_root() (sympy.liealgebras.type_b.TypeB method), 707

simple_root() (sympy.liealgebras.type_c.TypeC method), 709

simple_root() (sympy.liealgebras.type_d.TypeD method), 710

simple_root() (sympy.liealgebras.type_e.TypeE method), 712

simple_root() (sympy.liealgebras.type_f.TypeF method), 713

simple_root() (sympy.liealgebras.type_g.TypeG method), 714

simple_roots() (sympy.liealgebras.root_system.RootSystem method), 705

simpleDE() (in module sympy.series.formal), 1127

SimpleDomain (class in `sympy.polys.domains.simplesdomain`), 910

`simplify()` (in module `sympy.simplify.simplify`), 1154

`simplify()` (`sympy.core.expr.Expr` method), 132

`simplify()` (`sympy.matrices.dense.MutableDenseMatrix` method), 766

`simplify()` (`sympy.physics.vector.dyadic.Dyadic` method), 1563

`simplify()` (`sympy.physics.vector.vector.Vector` method), 1560

`simplify_gpgp()` (in module `sympy.physics.hep.gamma_matrices`), 1516

`simplify_index_permutations()` (in module `sympy.physics.secondquant`), 1496

`simplify_logic()` (in module `sympy.logic.boolalg`), 727

`sin` (class in `sympy.functions.elementary.trigonometric`), 432

`sinc` (class in `sympy.functions.elementary.trigonometric`), 437

`sine_transform()` (in module `sympy.integrals.transforms`), 670

`SingleDomain` (class in `sympy.stats.rv`), 1228

`SinglePSpace` (class in `sympy.stats.rv`), 1228

`SingletonRegistry` (class in `sympy.core.singleton`), 110

`singular` (`sympy.assumptions.ask.AssumptionKeys` attribute), 1088

`singularities()` (in module `sympy.calculus.singularities`), 1463

`SingularityFunction` (class in `sympy.functions.special.singularity_functions`), 481

`sinh` (class in `sympy.functions.elementary.hyperbolic`), 443

`size` (`sympy.combinatorics.permutations.Permutation` attribute), 228

`size` (`sympy.combinatorics.polyhedron.Polyhedron` attribute), 263

`size` (`sympy.combinatorics.prufer.Prufer` attribute), 266

`size` (`sympy.combinatorics.subsets.Subset` attribute), 272

`size()` (in module `sympy.codegen.fnodes`), 420

`sizeof()` (in module `sympy.codegen.cnodes`), 414

`SKIP()` (in module `sympy.utilities.pytest`), 1445

`skip()` (`sympy.combinatorics.graycode.GrayCode` method), 276

`slice()` (`sympy.polys.polyclasses.DMP` method), 922

`slice()` (`sympy.polys.polytools.Poly` method), 856

`slope` (`sympy.geometry.line.LinearEntity2D` attribute), 598

`slope()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1730

`smallest_angle_between()` (`sympy.geometry.line.LinearEntity` method), 589

`smoothness()` (in module `sympy.ntheory.factor_`), 306

`smoothness_p()` (in module `sympy.ntheory.factor_`), 307

`solve()` (in module `sympy.solvers.solvers`), 1296

`solve()` (`sympy.matrices.matrices.MatrixBase` method), 752

`solve()` (`sympy.matrices.sparse.SparseMatrix` method), 771

`solve_congruence()` (in module `sympy.ntheory.modular`), 323

`solve_de()` (in module `sympy.series.formal`), 1129

`solve_for_reaction_loads()` (`sympy.physics.continuum_mechanics.beam.Beam` method), 1731

`solve_least_squares()` (`sympy.matrices.matrices.MatrixBase` method), 752

`solve_least_squares()` (`sympy.matrices.sparse.SparseMatrix` method), 771

`solve_linear()` (in module `sympy.solvers.solvers`), 1303

`solve_linear_system()` (in module `sympy.solvers.solvers`), 1305

`solve_linear_system_LU()` (in module `sympy.solvers.solvers`), 1306

`solve_multipliers()` (`sympy.physics.mechanics.lagrange.Lagrangian` method), 1637

`solve_poly_inequality()` (in module `sympy.solvers.inequalities`), 1339

`solve_poly_system()` (in module `sympy.solvers.polysys`), 1313

`solve_rational_inequalities()` (in module `sympy.solvers.inequalities`), 1339

`solve_triangularized()` (in module `sympy.solvers.polysys`), 1313

`solve_undetermined_coeffs()` (in module `sympy.solvers.solvers`), 1306

`solve_univariate_inequality()` (in module `sympy.solvers.inequalities`), 1341

- [solveset\(\)](#) (in module `sympy.solvers.solveset`), [1350](#)
[solveset_complex\(\)](#) (in module `sympy.solvers.solveset`), [1351](#)
[solveset_real\(\)](#) (in module `sympy.solvers.solveset`), [1351](#)
[SOPform\(\)](#) (in module `sympy.logic.boolalg`), [718](#)
[sort_dims\(\)](#) (`sympy.physics.units.dimensions.DimensionSystem` static method), [1511](#)
[sort_key\(\)](#) (`sympy.combinatorics.partitions.Partition` method), [204](#)
[sort_key\(\)](#) (`sympy.core.basic.Basic` method), [106](#)
[sorted_components\(\)](#) (`sympy.tensor.tensor.TensMul` method), [1389](#)
[source](#) (`sympy.geometry.line.Ray` attribute), [594](#)
[source\(\)](#) (in module `sympy.utilities.source`), [1454](#)
[SpaceOrienter](#) (class in `sympy.vector.orienters`), [1800](#)
[SparseMatrix](#) (class in `sympy.matrices.sparse`), [767](#)
[speed](#) (`sympy.physics.optics.medium.Medium` attribute), [1715](#)
[speed](#) (`sympy.physics.optics.waves.TWave` attribute), [1721](#)
[speeds](#) (`sympy.physics.mechanics.system.SymbolicSystem` attribute), [1641](#)
[spin\(\)](#) (`sympy.geometry.polygon.RegularPolygon` method), [641](#)
[split\(\)](#) (`sympy.tensor.tensor.TensMul` method), [1389](#)
[split_list\(\)](#) (in module `sympy.utilities.runtests`), [1450](#)
[split_super_sub\(\)](#) (in module `sympy.printing.conventions`), [1053](#)
[split_symbols\(\)](#) (in module `sympy.parsing.sympy_parser`), [1456](#)
[split_symbols_custom\(\)](#) (in module `sympy.parsing.sympy_parser`), [1456](#)
[spoly\(\)](#) (in module `sympy.polys.groebnertools`), [997](#)
[sqf\(\)](#) (in module `sympy.polys.polytools`), [816](#)
[sqf_list\(\)](#) (in module `sympy.polys.polytools`), [816](#)
[sqf_list\(\)](#) (`sympy.polys.polyclasses.DMP` method), [922](#)
[sqf_list\(\)](#) (`sympy.polys.polytools.Poly` method), [856](#)
[sqf_list_include\(\)](#) (`sympy.polys.polyclasses.DMP` method), [922](#)
[sqf_list_include\(\)](#) (`sympy.polys.polytools.Poly` method), [856](#)
[sqf_norm\(\)](#) (in module `sympy.polys.polytools`), [815](#)
[sqf_norm\(\)](#) (`sympy.polys.polyclasses.DMP` method), [923](#)
[sqf_norm\(\)](#) (`sympy.polys.polytools.Poly` method), [857](#)
[sqf_norm_sys\(\)](#) (in module `sympy.solvers.diophantine`), [1338](#)
[sqf_part\(\)](#) (in module `sympy.polys.polytools`), [816](#)
[sqf_part\(\)](#) (`sympy.polys.polyclasses.DMP` method), [923](#)
[sqf_part\(\)](#) (`sympy.polys.polytools.Poly` method), [857](#)
[sqr\(\)](#) (`sympy.polys.polyclasses.DMP` method), [923](#)
[sqr\(\)](#) (`sympy.polys.polytools.Poly` method), [857](#)
[Sqrt](#) (class in `sympy.codegen.cfunctions`), [410](#)
[sqrt\(\)](#) (in module `sympy.functions.elementary.miscellaneous`), [456](#)
[sqrt\(\)](#) (`sympy.polys.domains.domain.Domain` method), [908](#)
[sqrt_mod\(\)](#) (in module `sympy.ntheory.residue_ntheory`), [327](#)
[sqrtdenest\(\)](#) (in module `sympy.simplify.sqrtdenest`), [1173](#)
[square](#) (`sympy.assumptions.ask.AssumptionKeys` attribute), [1088](#)
[square\(\)](#) (`sympy.polys.rings.PolyElement` method), [978](#)
[square_factor\(\)](#) (in module `sympy.solvers.diophantine`), [1328](#)
[srepr\(\)](#) (in module `sympy.printing.repr`), [1050](#)
[sring\(\)](#) (in module `sympy.polys.rings`), [972](#)
[sstrrepr\(\)](#) (in module `sympy.printing.str`), [1050](#)
[stabilizer\(\)](#) (`sympy.combinatorics.perm_groups.PermutationGroup` method), [257](#)
[stack\(\)](#) (`sympy.printing.pretty.stringpict.stringPict` static method), [1057](#)
[Standard_Cartan](#) (class in `sympy.liealgebras.cartan_type`), [716](#)
[standard_transformations](#) (in module `sympy.parsing.sympy_parser`), [1456](#)
[start](#) (`sympy.series.sequences.SeqBase` attribute), [1116](#)
[start](#) (`sympy.sets.sets.Interval` attribute), [1140](#)

State (class in sympy.physics.quantum.state), 1681
 state() (sympy.physics.secondquant.FixedBosonicBasis method), 1488
 state() (sympy.physics.secondquant.VarBosonicBasis method), 1487
 state_to_operators() (in module sympy.physics.quantum.operatorset), 1664
 StateBase (class in sympy.physics.quantum.state), 1681
 states (sympy.physics.mechanics.system.SymbolicSystem attribute), 1641
 std() (in module sympy.stats), 1227
 stieltjes (class in sympy.functions.special.zeta_functions), 529
 stirling() (in module sympy.functions.combinatorial.numbers), 473
 stop (sympy.series.sequences.SeqBase attribute), 1116
 Stream (class in sympy.codegen.ast), 403
 StrictGreaterThan (class in sympy.core.relational), 169
 StrictLessThan (class in sympy.core.relational), 172
 strides (sympy.tensor.indexed.IndexedBase attribute), 1372
 String (class in sympy.codegen.ast), 403
 stringify_expr() (in module sympy.parsing.sympy_parser), 1456
 stringPict (class in sympy.printing.pretty.stringpict), 1056
 strip_zero() (sympy.polys.rings.PolyElement method), 979
 strong_gens (sympy.combinatorics.perm_groups.PermutationGroup attribute), 258
 strong_presentation() (sympy.combinatorics.perm_groups.PermutationGroup method), 258
 StrPrinter (class in sympy.printing.str), 1050
 struct (class in sympy.codegen.cnodes), 414
 StudentT() (in module sympy.stats), 1214
 sturm() (in module sympy.polys.polytools), 815
 sturm() (sympy.polys.polyclasses.DMP method), 923
 sturm() (sympy.polys.polytools.Poly method), 858
 sub (in module sympy.printing.pretty.pretty_symbolic), 1055
 sub() (sympy.polys.domains.domain.Domain method), 908
 sub() (sympy.polys.polyclasses.DMF method), 924
 sub() (sympy.polys.polyclasses.DMP method), 923
 sub() (sympy.polys.polytools.Poly method), 858
 sub_ground() (sympy.polys.polyclasses.DMP method), 923
 sub_ground() (sympy.polys.polytools.Poly method), 858
 subdiagram_from_objects() (sympy.categories.Diagram method), 1744
 subfactorial (class in sympy.functions.combinatorial.factorials), 466
 subgroup() (sympy.combinatorics.perm_groups.PermutationGroup method), 258
 subgroup_search() (sympy.combinatorics.perm_groups.PermutationGroup method), 258
 SubModule (class in sympy.polys.agca.modules), 889
 submodule() (sympy.polys.agca.modules.Module method), 887
 submodule() (sympy.polys.agca.modules.QuotientModule method), 897
 submodule() (sympy.polys.agca.modules.SubModule method), 892
 SubQuotientModule (class in sympy.polys.agca.modules), 897
 subresultants() (in module sympy.polys.polytools), 807
 subresultants() (sympy.polys.polyclasses.DMP method), 923
 subresultants() (sympy.polys.polytools.Poly method), 858
 SubroutineCall (class in sympy.codegen.fnodes), 417
 SubroutineCall (class in sympy.codegen.fnodes), 417
 Subs (class in sympy.core.function), 185
 subs() (sympy.core.basic.Basic method), 107
 subs() (sympy.physics.vector.dyadic.Dyadic method), 1563
 subs() (sympy.physics.vector.vector.Vector method), 1560
 Subset (class in sympy.combinatorics.subsets), 268
 subset (sympy.combinatorics.subsets.Subset attribute), 272
 subset() (sympy.polys.agca.ideals.Ideal method), 895
 subset() (sympy.polys.agca.modules.Module method), 887

subset_from_bitlist() (sympy.combinatorics.subsets.Subset class method), 272
 subset_indices() (sympy.combinatorics.subsets.Subset class method), 273
 subsets() (in module sympy.utilities.iterables), 1434
 SubsSet (class in sympy.series.gruntz), 1109
 substitute_dummies() (in module sympy.physics.secondquant), 1495
 substitute_indices() (sympy.tensor.tensor.TensorAdd method), 1386
 Sum (class in sympy.concrete.summations), 361
 sum_of_four_squares() (in module sympy.solvers.diophantine), 1332
 sum_of_powers() (in module sympy.solvers.diophantine), 1332
 sum_of_squares() (in module sympy.solvers.diophantine), 1333
 sum_of_three_squares() (in module sympy.solvers.diophantine), 1331
 SumApprox (class in sympy.codegen.approximations), 392
 summation() (in module sympy.concrete.summations), 371
 sup (in module sympy.printing.pretty.pretty_symbology), 1055
 sup (sympy.sets.sets.Set attribute), 1136
 superposition_basis() (in module sympy.physics.quantum.grover), 1694
 superset (sympy.combinatorics.subsets.Subset attribute), 273
 superset_size (sympy.combinatorics.subsets.Subset attribute), 273
 support() (sympy.combinatorics.permutations.Permutation method), 228
 SurfaceBaseSeries (class in sympy.plotting.plot), 1072
 SurfaceOver2DRangeSeries (class in sympy.plotting.plot), 1072
 SWAP (in module sympy.physics.quantum.gate), 1692
 SwapGate (class in sympy.physics.quantum.gate), 1692
 swinnerton_dyer_poly() (in module sympy.polys.specialpolys), 872
 sylow_subgroup() (sympy.combinatorics.permutation_groups.PermutationGroup method), 260
 symarray() (in module sympy.matrices.dense), 759
 symb_2txt (in module sympy.printing.pretty.pretty_symbology), 1055
 Symbol (class in sympy.core.symbol), 133
 symbol (sympy.physics.secondquant.AntiSymmetricTensor attribute), 1495
 Symbol() (sympy.assumptions.handlers.calculus.AskFiniteH static method), 1097
 SymbolicSystem (class in sympy.physics.mechanics.system), 1637
 symbols() (in module sympy.core.symbol), 135
 symmetric (sympy.assumptions.ask.AssumptionKeys attribute), 1088
 symmetric() (sympy.combinatorics.generators method), 231
 symmetric_difference() (sympy.sets.sets.Set method), 1136
 symmetric_poly() (in module sympy.polys.specialpolys), 873
 symmetric_residue() (in module sympy.ntheory.modular), 322
 SymmetricGroup() (in module sympy.combinatorics.named_groups), 279
 symmetrize() (in module sympy.polys.polyfuncs), 864
 sympify() (in module sympy.core.sympify), 91
 sympy (module), 776
 sympy.assumptions (module), 1076
 sympy.assumptions.ask (module), 1076
 sympy.assumptions.assume (module), 1092
 sympy.assumptions.handlers (module), 1095
 sympy.assumptions.handlers.calculus (module), 1095
 sympy.assumptions.handlers.nttheory (module), 1097
 sympy.assumptions.handlers.order (module), 1097
 sympy.assumptions.handlers.sets (module), 1098
 sympy.assumptions.refine (module), 1094
 sympy.calculus (module), 1460
 sympy.calculus.euler (module), 1460
 sympy.calculus.finite_diff (module), 1463
 sympy.calculus.singularities (module), 1461
 sympy.categories (module), 1736
 sympy.categories.diagram_drawing (module), 1736
 sympy.categories.permutation_groups (module), 260
 sympy.codegen.algorithms (module), 421
 sympy.codegen.approximations (module), 391
 sympy.codegen.ast (module), 393

`sympy.codegen.cfunctions` (module), 409
`sympy.codegen.cnodes` (module), 413
`sympy.codegen.cutils` (module), 423
`sympy.codegen.cxxnodes` (module), 414
`sympy.codegen.fnodes` (module), 414
`sympy.codegen.futils` (module), 423
`sympy.codegen.pyutils` (module), 422
`sympy.codegen.rewriting` (module), 389
`sympy.combinatorics.generators` (module), 231
`sympy.combinatorics.graycode` (module), 274
`sympy.combinatorics.group_constructs` (module), 287
`sympy.combinatorics.named_groups` (module), 279
`sympy.combinatorics.partitions` (module), 202
`sympy.combinatorics.perm_groups` (module), 232
`sympy.combinatorics.permutations` (module), 207
`sympy.combinatorics.polyhedron` (module), 261
`sympy.combinatorics.prufer` (module), 264
`sympy.combinatorics.subsets` (module), 268
`sympy.combinatorics.tensor_can` (module), 289
`sympy.combinatorics.testutil` (module), 287
`sympy.combinatorics.util` (module), 281
`sympy.core.add` (module), 158
`sympy.core.assumptions` (module), 94
`sympy.core.basic` (module), 96
`sympy.core.cache` (module), 96
`sympy.core.compatibility` (module), 198
`sympy.core.containers` (module), 197
`sympy.core.core` (module), 110
`sympy.core.evalf` (module), 196
`sympy.core.expr` (module), 111
`sympy.core.exprtools` (module), 200
`sympy.core.function` (module), 176
`sympy.core.mod` (module), 161
`sympy.core.mul` (module), 155
`sympy.core.multidimensional` (module), 175
`sympy.core.numbers` (module), 138
`sympy.core.power` (module), 152
`sympy.core.relational` (module), 161
`sympy.core.singleton` (module), 110
`sympy.core.symbol` (module), 133
`sympy.core.sympify` (module), 91
`sympy.crypto.crypto` (module), 336
`sympy.diffgeom` (module), 1753
`sympy.functions` (module), 425
`sympy.functions.special.bessel` (module), 512
`sympy.functions.special.beta_functions` (module), 490
`sympy.functions.special.elliptic_integrals` (module), 534
`sympy.functions.special.error_functions` (module), 491
`sympy.functions.special.gamma_functions` (module), 483
`sympy.functions.special.mathieu_functions` (module), 537
`sympy.functions.special.polynomials` (module), 540
`sympy.functions.special.singularity_functions` (module), 481
`sympy.functions.special.zeta_functions` (module), 524
`sympy.geometry.curve` (module), 608
`sympy.geometry.ellipse` (module), 612
`sympy.geometry.entity` (module), 561
`sympy.geometry.line` (module), 579
`sympy.geometry.plane` (module), 650
`sympy.geometry.point` (module), 567
`sympy.geometry.polygon` (module), 627
`sympy.geometry.util` (module), 564
`sympy.holonomic` (module), 657
`sympy.holonomic.holonomic` (module), 658, 659, 664, 665
`sympy.integrals` (module), 666
`sympy.integrals.intpoly` (module), 700
`sympy.integrals.meijerint_doc` (module), 684
`sympy.integrals.transforms` (module), 667
`sympy.liealgebras` (module), 703
`sympy.logic` (module), 717
`sympy.matrices` (module), 729
`sympy.matrices.expressions` (module), 778
`sympy.matrices.expressions.blockmatrix` (module), 784
`sympy.matrices.immutable` (module), 777
`sympy.matrices.matrices` (module), 729
`sympy.matrices.sparse` (module), 767
`sympy.ntheory.continued_fraction` (module), 330
`sympy.ntheory.egyptian_fraction` (module), 334
`sympy.ntheory.factor_` (module), 306
`sympy.ntheory.generate` (module), 298
`sympy.ntheory.modular` (module), 322
`sympy.ntheory.multinomial` (module), 324
`sympy.ntheory.partitions_` (module), 325
`sympy.ntheory.primetest` (module), 325
`sympy.ntheory.residue_ntheory` (module), 326
`sympy.physics` (module), 1469
`sympy.physics.continuum_mechanics.beam` (module), 1722

- `sympy.physics.hep.gamma_matrices` (module), 1514
- `sympy.physics.hydrogen` (module), 1469
- `sympy.physics.matrices` (module), 1472
- `sympy.physics.mechanics.body` (module), 1644
- `sympy.physics.mechanics.kane` (module), 1631
- `sympy.physics.mechanics.lagrange` (module), 1634
- `sympy.physics.mechanics.linearize` (module), 1641
- `sympy.physics.mechanics.particle` (module), 1621
- `sympy.physics.mechanics.rigidbody` (module), 1623
- `sympy.physics.mechanics.system` (module), 1637
- `sympy.physics.optics.gaussopt` (module), 1702
- `sympy.physics.optics.medium` (module), 1713
- `sympy.physics.optics.utils` (module), 1715
- `sympy.physics.optics.waves` (module), 1719
- `sympy.physics.paulialgebra` (module), 1474
- `sympy.physics.qho_1d` (module), 1474
- `sympy.physics.quantum.anticommutator` (module), 1647
- `sympy.physics.quantum.cartesian` (module), 1656
- `sympy.physics.quantum.cg` (module), 1648
- `sympy.physics.quantum.circuitplot` (module), 1688
- `sympy.physics.quantum.commutator` (module), 1650
- `sympy.physics.quantum.constants` (module), 1651
- `sympy.physics.quantum.dagger` (module), 1652
- `sympy.physics.quantum.gate` (module), 1689
- `sympy.physics.quantum.grover` (module), 1694
- `sympy.physics.quantum.hilbert` (module), 1657
- `sympy.physics.quantum.innerproduct` (module), 1653
- `sympy.physics.quantum.operator` (module), 1658
- `sympy.physics.quantum.operatorset` (module), 1663
- `sympy.physics.quantum.piab` (module), 1701
- `sympy.physics.quantum.qapply` (module), 1665
- `sympy.physics.quantum.qft` (module), 1696
- `sympy.physics.quantum.qubit` (module), 1696
- `sympy.physics.quantum.represent` (module), 1666
- `sympy.physics.quantum.shor` (module), 1701
- `sympy.physics.quantum.spin` (module), 1670
- `sympy.physics.quantum.state` (module), 1680
- `sympy.physics.quantum.tensorproduct` (module), 1654
- `sympy.physics.secondquant` (module), 1476
- `sympy.physics.sho` (module), 1475
- `sympy.physics.units.dimensions` (module), 1509
- `sympy.physics.units.prefixes` (module), 1512
- `sympy.physics.units.quantities` (module), 1513
- `sympy.physics.units.unitsystem` (module), 1512
- `sympy.physics.units.util` (module), 1513
- `sympy.physics.vector.functions` (module), 1569
- `sympy.physics.vector.point` (module), 1564
- `sympy.physics.wigner` (module), 1497
- `sympy.plotting.plot` (module), 1059
- `sympy.plotting.pygletplot` (module), 1072
- `sympy.printing.ccode` (module), 1024
- `sympy.printing.codeprinter` (module), 1053
- `sympy.printing.conventions` (module), 1053
- `sympy.printing.cxxcode` (module), 1027
- `sympy.printing.fcode` (module), 1030
- `sympy.printing.gtk` (module), 1044
- `sympy.printing.jscode` (module), 1035
- `sympy.printing.julia` (module), 1037
- `sympy.printing.lambdarepr` (module), 1045
- `sympy.printing.latex` (module), 1045
- `sympy.printing.mathematica` (module), 1034
- `sympy.printing.mathml` (module), 1047
- `sympy.printing.octave` (module), 1039
- `sympy.printing.precedence` (module), 1054
- `sympy.printing.pretty.pretty` (module), 1023
- `sympy.printing.pretty.pretty_symbolology` (module), 1055
- `sympy.printing.pretty.stringpict` (module), 1056
- `sympy.printing.preview` (module), 1052
- `sympy.printing.printer` (module), 1019
- `sympy.printing.pycode` (module), 1048
- `sympy.printing.python` (module), 1049
- `sympy.printing.rcode` (module), 1028
- `sympy.printing.repr` (module), 1049
- `sympy.printing.rust` (module), 1042
- `sympy.printing.str` (module), 1050
- `sympy.printing.theanocode` (module), 1044
- `sympy.printing.tree` (module), 1050
- `sympy.sets.fancysets` (module), 1146
- `sympy.sets.sets` (module), 1132

`sympy.simplify.combsimp` (module), 1173
`sympy.simplify.cse_main` (module), 1103, 1174
`sympy.simplify.epathtools` (module), 1177
`sympy.simplify.hyperexpand` (module), 1176
`sympy.simplify.hyperexpand_doc` (module), 1187
`sympy.simplify.powsimp` (module), 1170
`sympy.simplify.radsimp` (module), 1162
`sympy.simplify.ratsimp` (module), 1169
`sympy.simplify.simplify` (module), 77
`sympy.simplify.sqrtdenest` (module), 1173
`sympy.simplify.traversaltools` (module), 1177
`sympy.simplify.trigsimp` (module), 1169
`sympy.solvers` (module), 1295
`sympy.solvers.inequalities` (module), 1339
`sympy.solvers.ode` (module), 1283
`sympy.solvers.pde` (module), 1295
`sympy.solvers.recurr` (module), 1310
`sympy.solvers.solve` (module), 1342
`sympy.stats` (module), 1189
`sympy.stats.crv` (module), 1229
`sympy.stats.crv_types` (module), 1229
`sympy.stats.Die()` (in module `sympy.stats.crv_types`), 1229
`sympy.stats.frv` (module), 1229
`sympy.stats.frv_types` (module), 1229
`sympy.stats.Normal()` (in module `sympy.stats.crv_types`), 1229
`sympy.stats.rv` (module), 1228
`sympy.tensor` (module), 1360
`sympy.tensor.array` (module), 1360
`sympy.tensor.index_methods` (module), 1372
`sympy.tensor.indexed` (module), 1366
`sympy.tensor.tensor` (module), 1375
`sympy.utilities` (module), 1390
`sympy.utilities.autowrap` (module), 1391
`sympy.utilities.codegen` (module), 1397
`sympy.utilities.decorator` (module), 1409
`sympy.utilities.enumerative` (module), 1410
`sympy.utilities.iterables` (module), 1417
`sympy.utilities.lambdify` (module), 1437
`sympy.utilities.memoization` (module), 1441
`sympy.utilities.misc` (module), 1441
`sympy.utilities.pkgdata` (module), 1444
`sympy.utilities.pytest` (module), 1445
`sympy.utilities.randtest` (module), 1446
`sympy.utilities.runtests` (module), 1447
`sympy.utilities.source` (module), 1454
`sympy.utilities.timeutils` (module), 1454
`SymPyDocTestFinder` (class in `sympy.utilities.runtests`), 1447
`SymPyDocTestRunner` (class in `sympy.utilities.runtests`), 1447

`SymPyOutputChecker` (class in `sympy.utilities.runtests`), 1448
`sympytestfile()` (in module `sympy.utilities.runtests`), 1450
`SymPyTestResults` (in module `sympy.utilities.runtests`), 1448
`syzygy_module()` (`sympy.polys.agca.modules.SubModule` method), 892

T

`T` (in module `sympy.physics.quantum.gate`), 1692
`T` (`sympy.matrices.expressions.MatrixExpr` attribute), 779
`t` (`sympy.physics.quantum.shor.CMod` attribute), 1701
`table()` (`sympy.matrices.matrices.MatrixBase` method), 753
`tail_degree()` (`sympy.polys.rings.PolyElement` method), 979
`tail_degrees()` (`sympy.polys.rings.PolyElement` method), 979
`take()` (in module `sympy.utilities.iterables`), 1435
`tan` (class in `sympy.functions.elementary.trigonometric`), 434
`tangent_lines()` (`sympy.geometry.ellipse.Ellipse` method), 623
`tanh` (class in `sympy.functions.elementary.hyperbolic`), 444
`targets` (`sympy.physics.quantum.gate.CGate` attribute), 1690
`targets` (`sympy.physics.quantum.gate.CNotGate` attribute), 1692
`targets` (`sympy.physics.quantum.gate.Gate` attribute), 1690
`targets` (`sympy.physics.quantum.gate.UGate` attribute), 1690
`targets` (`sympy.physics.quantum.grover.OracleGate` attribute), 1694
`taxicab_distance()` (`sympy.geometry.point.Point` method), 573
`taylor_term()` (`sympy.core.expr.Expr` method), 132
`TC()` (`sympy.polys.polyclasses.ANP` method), 924
`TC()` (`sympy.polys.polyclasses.DMP` method), 918
`TC()` (`sympy.polys.polytools.Poly` method), 823
in `TensAdd` (class in `sympy.tensor.tensor`), 1385
in `TensExpr` (class in `sympy.tensor.tensor`), 1383
in `TensMul` (class in `sympy.tensor.tensor`), 1387

tensor_indices() (in module sympy.tensor.tensor), 1379	ThinLens (class in sympy.physics.optics.gaussopt), 1706
tensor_mul() (in module sympy.tensor.tensor), 1389	threaded() (in module sympy.utilities.decorator), 1410
tensor_product_simp() (in module sympy.physics.quantum.tensorproduct), 1655	threaded_factory() (in module sympy.utilities.decorator), 1410
tensorcontraction() (in module sympy.tensor.array), 1364	time (sympy.physics.quantum.state.TimeDepState attribute), 1683
TensorHead (class in sympy.tensor.tensor), 1381	time_derivative() (in module sympy.physics.vector.functions), 1578
TensorIndex (class in sympy.tensor.tensor), 1378	time_period (sympy.physics.optics.waves.TWave attribute), 1721
TensorIndexType (class in sympy.tensor.tensor), 1376	timed() (in module sympy.utilities.timeutils), 1454
TensorProduct (class in sympy.diffgeom), 1760	TimeDepBra (class in sympy.physics.quantum.state), 1683
TensorProduct (class in sympy.physics.quantum.tensorproduct), 1654	TimeDepKet (class in sympy.physics.quantum.state), 1684
tensorproduct() (in module sympy.tensor.array), 1365	TimeDepState (class in sympy.physics.quantum.state), 1683
TensorSymmetry (class in sympy.tensor.tensor), 1379	tiny (sympy.codegen.ast.FloatType attribute), 400
tensorsymmetry() (in module sympy.tensor.tensor), 1380	to_algebraic_integer() (sympy.polys.numberfields.AlgebraicNumber method), 868
TensorType (class in sympy.tensor.tensor), 1380	to_cnf() (in module sympy.logic.boolalg), 726
terminal_width() (sympy.printing.pretty.stringpict.string method), 1057	to_cnf() (sympy.polys.polyclasses.ANP method), 925
terms() (sympy.polys.polyclasses.DMP method), 923	to_dict() (sympy.polys.polyclasses.DMP method), 923
terms() (sympy.polys.polytools.Poly method), 859	to_dnf() (in module sympy.logic.boolalg), 726
terms() (sympy.polys.rings.PolyElement method), 979	to_exact() (sympy.polys.polyclasses.DMP method), 923
terms_gcd() (in module sympy.polys.polytools), 811	to_exact() (sympy.polys.polytools.Poly method), 860
terms_gcd() (sympy.polys.polyclasses.DMP method), 923	to_expr() (sympy.holonomic.holonomic.HolonomicFunction method), 663
terms_gcd() (sympy.polys.polytools.Poly method), 859	to_field() (sympy.polys.polyclasses.DMP method), 923
termwise() (sympy.polys.polytools.Poly method), 859	to_field() (sympy.polys.polytools.Poly method), 860
test() (in module sympy.utilities.runtests), 1452	to_hyper() (sympy.holonomic.holonomic.HolonomicFunction method), 662
test_derivative_numerically() (in module sympy.utilities.randtest), 1446	to_linearizer() (sympy.physics.mechanics.kane.KanesMethod method), 1634
TGate (class in sympy.physics.quantum.gate), 1691	to_linearizer() (sympy.physics.mechanics.lagrange.Lagrangian method), 1637
theano_function() (in module sympy.printing.theanocode), 1044	to_list() (sympy.polys.polyclasses.ANP method), 925
TheanoPrinter (class in sympy.printing.theanocode), 1044	to_list() (sympy.polys.polyclasses.DMP method), 923

`to_matrix()` (sympy.physics.vector.dyadic.DyadicToken (class in sympy.codegen.ast), 404
method), 1563
`to_matrix()` (sympy.physics.vector.vector.Vector
method), 1560
`to_matrix()` (sympy.vector.dyadic.Dyadic
method), 1796
`to_matrix()` (sympy.vector.vector.Vector
method), 1795
`to_meijerg()` (sympy.holonomic.holonomic.HolonomicFunction), 860
method), 663
`to_number_field()` (in module
sympy.polys.numberfields), 867
`to_prufer()` (sympy.combinatorics.prufer.Prufer
static method), 266
`to_rational()` (sympy.polys.domains.RealField
method), 916
`to_ring()` (sympy.polys.polyclasses.DMP
method), 923
`to_ring()` (sympy.polys.polytools.Poly
method), 860
`to_sequence()` (sympy.holonomic.holonomic.HolonomicFunction), 575
method), 660
`to_sympy()` (sympy.polys.domains.AlgebraicField
method), 914
`to_sympy()` (sympy.polys.domains.domain.Domain
method), 908
`to_sympy()` (sympy.polys.domains.ExpressionDomain
method), 917
`to_sympy()` (sympy.polys.domains.FiniteField
method), 912
`to_sympy()` (sympy.polys.domains.FractionField
method), 916
`to_sympy()` (sympy.polys.domains.PolynomialRing
method), 913
`to_sympy()` (sympy.polys.domains.RealField
method), 916
`to_sympy_dict()` (sympy.polys.polyclasses.ANP
method), 925
`to_sympy_dict()` (sympy.polys.polyclasses.DMP
method), 923
`to_sympy_list()` (sympy.polys.polyclasses.ANP
method), 925
`to_sympy_list()` (sympy.polys.polyclasses.DMP
method), 923
`to_tree()` (sympy.combinatorics.prufer.Prufer
static method), 267
`to_tuple()` (sympy.polys.polyclasses.ANP
method), 925
`to_tuple()` (sympy.polys.polyclasses.DMP
method), 923
`together()` (in module
sympy.polys.rationaltools), 875
`together()` (sympy.core.expr.Expr method),
132
`topological_sort()` (in module
sympy.utilities.iterables), 1435
`topological_sort()` (sympy.codegen.ast.CodeBlock
class method), 397
`total_degree()` (sympy.polys.polyclasses.DMP
method), 923
`total_degree()` (sympy.polys.polytools.Poly
method), 860
`totient` (class in sympy.ntheory.factor_), 317
`totientrange()` (sympy.ntheory.generate.Sieve
method), 300
`Trace` (class in sympy.matrices.expressions),
782
`trailing()` (in module sympy.ntheory.factor_),
308
`transcendental`, 95
`transcendental` (sympy.assumptions.ask.AssumptionKeys
attribute), 1089
`transform()` (sympy.geometry.point.Point2D
method), 575
`transform()` (sympy.geometry.point.Point3D
method), 578
`transform()` (sympy.integrals.Integral
method), 692
`transform()` (sympy.polys.polyclasses.DMP
method), 923
`transform()` (sympy.polys.polytools.Poly
method), 860
`transformation_to_DN()` (in module
sympy.solvers.diophantine), 1325
`transitivity_degree` (sympy.combinatorics.perm_groups.Perm
attribute), 260
`translate()` (in module sympy.utilities.misc),
1443
`translate()` (sympy.geometry.curve.Curve
method), 611
`translate()` (sympy.geometry.entity.GeometryEntity
method), 563
`translate()` (sympy.geometry.point.Point2D
method), 575
`translate()` (sympy.geometry.point.Point3D
method), 578
`Transpose` (class in
sympy.matrices.expressions), 782
`transpose()` (sympy.matrices.expressions.blockmatrix.BlockM
method), 784
`transpositions()` (sympy.combinatorics.permutations.Permut
method), 228
`transverse_magnification()` (in module
sympy.physics.optics.utils), 1719
`tree()` (in module sympy.printing.tree), 1051
`tree_cse()` (in module
sympy.simplify.cse_main), 1176

[tree_repr](#) (sympy.combinatorics.prufer.Prufer attribute), [267](#)
[Triangle](#) (class in sympy.geometry.polygon), [641](#)
[triangular](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1089](#)
[Triangular\(\)](#) (in module sympy.stats), [1215](#)
[trigamma\(\)](#) (in module sympy.functions.special.gamma_functions), [488](#)
[trigsimp\(\)](#) (in module sympy.simplify.trigsimp), [1169](#)
[trigsimp\(\)](#) (sympy.core.expr.Expr method), [132](#)
[trunc\(\)](#) (in module sympy.polys.polytools), [813](#)
[trunc\(\)](#) (sympy.polys.polyclasses.DMP method), [923](#)
[trunc\(\)](#) (sympy.polys.polytools.Poly method), [861](#)
[truncate\(\)](#) (sympy.series.formal.FormalPowerSeries method), [1124](#)
[truncate\(\)](#) (sympy.series.fourier.FourierSeries method), [1122](#)
[Tuple](#) (class in sympy.core.containers), [197](#)
[tuple_count\(\)](#) (sympy.core.containers.Tuple method), [197](#)
[TWave](#) (class in sympy.physics.optics.waves), [1719](#)
[twoform_to_matrix\(\)](#) (in module sympy.diffgeom), [1766](#)
[TwoQubitGate](#) (class in sympy.physics.quantum.gate), [1690](#)
[Type](#) (class in sympy.codegen.ast), [404](#)
[TypeA](#) (class in sympy.liealgebras.type_a), [705](#)
[TypeB](#) (class in sympy.liealgebras.type_b), [706](#)
[TypeC](#) (class in sympy.liealgebras.type_c), [708](#)
[TypeD](#) (class in sympy.liealgebras.type_d), [709](#)
[TypeE](#) (class in sympy.liealgebras.type_e), [711](#)
[TypeF](#) (class in sympy.liealgebras.type_f), [712](#)
[TypeG](#) (class in sympy.liealgebras.type_g), [713](#)

U

[U\(\)](#) (in module sympy.printing.pretty.pretty_symbol), [1055](#)
[udivisor_count\(\)](#) (in module sympy.ntheory.factor_), [316](#)
[udivisor_sigma](#) (class in sympy.ntheory.factor_), [319](#)
[udivisors\(\)](#) (in module sympy.ntheory.factor_), [315](#)
[Ufuncify\(\)](#) (in module sympy.utilities.autowrap), [1395](#)
[UfuncifyCodeWrapper](#) (class in sympy.utilities.autowrap), [1392](#)
[UGate](#) (class in sympy.physics.quantum.gate), [1690](#)
[uncouple\(\)](#) (in module sympy.physics.quantum.spin), [1679](#)
[Unequality](#) (class in sympy.core.relational), [169](#)
[UnevaluatedExpr](#) (class in sympy.core.expr), [132](#)
[unflatten\(\)](#) (in module sympy.utilities.iterables), [1436](#)
[UnificationFailed](#) (class in sympy.polys.polyerrors), [999](#)
[uninform\(\)](#) (in module sympy.stats), [1216](#)
[UniformSum\(\)](#) (in module sympy.stats), [1217](#)
[unify\(\)](#) (sympy.polys.domains.domain.Domain method), [908](#)
[unify\(\)](#) (sympy.polys.polyclasses.ANP method), [925](#)
[unify\(\)](#) (sympy.polys.polyclasses.DMP method), [923](#)
[unify\(\)](#) (sympy.polys.polytools.Poly method), [861](#)
[union](#) (class in sympy.codegen.cnodes), [414](#)
[Union](#) (class in sympy.sets.sets), [1141](#)
[union\(\)](#) (sympy.polys.agca.ideals.Ideal method), [895](#)
[union\(\)](#) (sympy.polys.agca.modules.SubModule method), [893](#)
[union\(\)](#) (sympy.series.gruntz.SubsSet method), [1110](#)
[union\(\)](#) (sympy.sets.sets.Set method), [1137](#)
[uniq\(\)](#) (in module sympy.utilities.iterables), [1436](#)
[unit](#) (sympy.geometry.point.Point attribute), [573](#)
[unit](#) (sympy.polys.polytools.Poly attribute), [861](#)
[unit_triangular](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1089](#)
[unitary](#) (sympy.assumptions.ask.AssumptionKeys attribute), [1090](#)
[UnitaryOperator](#) (class in sympy.physics.quantum.operator), [1660](#)
[UnitSystem](#) (class in sympy.physics.units.unitsystem),

- 1512
- UnivariatePolynomialError (class in sympy.polys.polyerrors), 1000
- UniversalSet (class in sympy.sets.sets), 1145
- unrank() (sympy.combinatorics.graycode.GrayCode class method), 277
- unrank() (sympy.combinatorics.prufer.Prufer class method), 267
- unrank_binary() (sympy.combinatorics.subsets.Subset class method), 273
- unrank_gray() (sympy.combinatorics.subsets.Subset class method), 274
- unrank_lex() (sympy.combinatorics.permutations.Permutation class method), 229
- unrank_nonlex() (sympy.combinatorics.permutations.Permutation class method), 229
- unrank_trotterjohnson() (sympy.combinatorics.permutations.Permutation class method), 229
- UnsignedIntType (class in sympy.codegen.ast), 406
- upper (sympy.physics.secondquant.AntiSymmetricTensor attribute), 1495
- upper (sympy.tensor.indexed.Idx attribute), 1368
- upper_triangular (sympy.assumptions.ask.AssumptionKey attribute), 1090
- upper_triangular_solve() (sympy.matrices.matrices.MatrixBase method), 753
- uppergamma (class in sympy.functions.special.gamma_functions), 488
- use (class in sympy.codegen.fnodes), 420
- use() (in module sympy.simplify.traversaltools), 1177
- use_rename (class in sympy.codegen.fnodes), 420
- using (class in sympy.codegen.cxxnodes), 414
- V**
- v1pt_theory() (sympy.physics.vector.point.Point method), 1568
- v2pt_theory() (sympy.physics.vector.point.Point method), 1568
- values() (sympy.core.containers.Dict method), 198
- var() (in module sympy.core.symbol), 137
- VarBosonicBasis (class in sympy.physics.secondquant), 1486
- Variable (class in sympy.codegen.ast), 406
- variable (sympy.physics.continuum_mechanics.Beam attribute), 1731
- variable_map() (sympy.physics.vector.frame.ReferenceFrame method), 1556
- variables (sympy.core.function.Lambda attribute), 176
- variables (sympy.core.function.Subs attribute), 186
- variables (sympy.physics.quantum.operator.DifferentialOperator attribute), 1663
- variables (sympy.physics.quantum.state.Wavefunction attribute), 1688
- variables (sympy.series.sequences.SeqBase attribute), 1116
- variables (sympy.utilities.codegen.Routine attribute), 1399
- variance (in module sympy.stats), 1225
- variance() (in module sympy.stats), 1224
- variations() (in module sympy.utilities.iterables), 1436
- vech() (sympy.matrices.matrices.MatrixBase method), 753
- Vector (class in sympy.physics.vector.vector), 1557
- Vector (class in sympy.vector.vector), 1793
- vectorize (class in sympy.core.multidimensional), 175
- vectorize_keybasis() (in module sympy.diffgeom), 1766
- vel() (sympy.physics.vector.point.Point method), 1569
- verify_numerically() (in module sympy.utilities.randtest), 1446
- vertices (sympy.combinatorics.polyhedron.Polyhedron attribute), 263
- vertices (sympy.geometry.polygon.Polygon attribute), 633
- vertices (sympy.geometry.polygon.RegularPolygon attribute), 641
- vertices (sympy.geometry.polygon.Triangle attribute), 649
- VF() (in module sympy.printing.pretty.pretty_symbology), 1055
- viete() (in module sympy.polys.polyfuncs), 865
- vlatex() (in module sympy.physics.vector.printing), 1575
- vobj() (in module sympy.printing.pretty.pretty_symbology), 1055
- VonMises() (in module sympy.stats), 1218
- vpprint() (in module sympy.physics.vector.printing), 1574
- vpprint() (in module sympy.physics.vector.printing), 1574

- vradius (sympy.geometry.ellipse.Circle attribute), 626
 vradius (sympy.geometry.ellipse.Ellipse attribute), 624
 vring() (in module sympy.polys.rings), 972
- ## W
- w (sympy.physics.optics.gaussopt.BeamParameter attribute), 1710
 w_0 (sympy.physics.optics.gaussopt.BeamParameter attribute), 1710
 waist2rayleigh() (in module sympy.physics.optics.gaussopt), 1710
 waist_approximation_limit (sympy.physics.optics.gaussopt.BeamParameter attribute), 1710
 Wavefunction (class in sympy.physics.quantum.state), 1684
 wavelength (sympy.physics.optics.waves.TWave attribute), 1722
 wavenumber (sympy.physics.optics.waves.TWave attribute), 1722
 WedgeProduct (class in sympy.diffgeom), 1761
 Weibull() (in module sympy.stats), 1219
 WeylGroup (class in sympy.liealgebras.weyl_group), 714
 WGate (class in sympy.physics.quantum.grover), 1694
 where() (in module sympy.stats), 1224
 While (class in sympy.codegen.ast), 408
 wicks() (in module sympy.physics.secondquant), 1490
 width (sympy.categories.diagram_drawing.Diagram attribute), 1747
 width() (sympy.printing.pretty.stringpict.stringPict method), 1057
 Wigner3j (class in sympy.physics.quantum.cg), 1649
 Wigner6j (class in sympy.physics.quantum.cg), 1649
 Wigner9j (class in sympy.physics.quantum.cg), 1649
 wigner_3j() (in module sympy.physics.wigner), 1500
 wigner_6j() (in module sympy.physics.wigner), 1502
 wigner_9j() (in module sympy.physics.wigner), 1503
 WignerD (class in sympy.physics.quantum.spin), 1672
 WignerSemicircle() (in module sympy.stats), 1219
 Wild (class in sympy.core.symbol), 133
 WildFunction (class in sympy.core.function), 177
 write() (sympy.utilities.codegen.CodeGen method), 1400
 write() (sympy.utilities.runtests.PyTestReporter method), 1447
 wronskian() (in module sympy.matrices.dense), 758
- ## X
- X (in module sympy.physics.quantum.gate), 1692
 x (sympy.geometry.point.Point2D attribute), 575
 x (sympy.geometry.point.Point3D attribute), 578
 x (sympy.physics.vector.frame.ReferenceFrame attribute), 1557
 XGate (class in sympy.physics.quantum.gate), 1691
 XKet (class in sympy.physics.quantum.cartesian), 1656
 xobj() (in module sympy.printing.pretty.pretty_symbology), 1055
 XOp (class in sympy.physics.quantum.cartesian), 1656
 XOR (class in sympy.logic.boolalg), 722
 xreplace() (sympy.core.basic.Basic method), 109
 xring() (in module sympy.polys.rings), 972
 xstr() (in module sympy.printing.pretty.pretty_symbology), 1055
 xsym() (in module sympy.printing.pretty.pretty_symbology), 1055
 xthreaded() (in module sympy.utilities.decorator), 1410
 xypic_draw_diagram() (in module sympy.categories.diagram_drawing), 1752
 XypicDiagramDrawer (class in sympy.categories.diagram_drawing), 1749

Y

Y (in module sympy.physics.quantum.gate),
[1692](#)

y (sympy.geometry.point.Point2D attribute),
[575](#)

y (sympy.geometry.point.Point3D attribute),
[579](#)

y (sympy.physics.vector.frame.ReferenceFrame
attribute), [1557](#)

ydirection (sympy.geometry.line.Ray2D at-
tribute), [602](#)

ydirection (sympy.geometry.line.Ray3D at-
tribute), [606](#)

YGate (class in sympy.physics.quantum.gate),
[1691](#)

yn (class in sympy.functions.special.bessel),
[516](#)

Ynm (class in sympy.functions.special.spherical_harmonics),
[550](#)

Ynm_c() (in module
sympy.functions.special.spherical_harmonics),
[552](#)

YOp (class in sympy.physics.quantum.cartesian),
[1656](#)

Z

Z (in module sympy.physics.quantum.gate),
[1692](#)

z (sympy.geometry.point.Point3D attribute),
[579](#)

z (sympy.physics.vector.frame.ReferenceFrame
attribute), [1557](#)

zdirection (sympy.geometry.line.Ray3D at-
tribute), [607](#)

zero, [95](#)

Zero (class in sympy.core.numbers), [144](#)

zero (sympy.assumptions.ask.AssumptionKeys
attribute), [1090](#)

zero (sympy.polys.polytools.Poly attribute),
[861](#)

ZeroMatrix (class in
sympy.matrices.expressions), [783](#)

zeros() (in module sympy.matrices.dense),
[755](#)

zeta (class in sympy.functions.special.zeta_functions),
[524](#)

ZGate (class in sympy.physics.quantum.gate),
[1691](#)

zip_row_op() (sympy.matrices.dense.MutableDenseMatrix
method), [766](#)

zip_row_op() (sympy.matrices.sparse.MutableSparseMatrix
method), [775](#)

Znm (class in sympy.functions.special.spherical_harmonics),
[552](#)

ZOp (class in sympy.physics.quantum.cartesian),
[1656](#)