SymPy Documentation

Release 0.7.3

SymPy Development Team

December 12, 2013

CONTENTS

1 Installation **1**
1.1 Source . . . 1
1.2 Git . . . 2
1.3 Anaconda . . . 2
1.4 Other Methods . . . 2
1.5 Run SymPy . . . 2
1.6 Questions . . . 3

2 SymPy Tutorial **5**
2.1 Preliminaries . . . 5
2.2 Introduction . . . 7
2.3 Printing . . . 10
2.4 Gotchas . . . 16
2.5 Basic Operations . . . 20
2.6 Simplification . . . 23
2.7 Calculus . . . 37
2.8 Solvers . . . 42
2.9 Matrices . . . 45
2.10 Advanced Expression Manipulation . . . 51

3 Gotchas and Pitfalls **57**
3.1 Introduction . . . 57
3.2 Equals Signs (=) . . . 57
3.3 Variables . . . 58
3.4 Symbolic Expressions . . . 61
3.5 Special Symbols . . . 65
3.6 Getting help from within SymPy . . . 68

4 SymPy User's Guide **71**
4.1 Introduction . . . 71
4.2 Learning SymPy . . . 71
4.3 SymPy's Architecture . . . 73
4.4 Contributing . . . 80

5 SymPy Modules Reference **81**
5.1 SymPy Core . . . 81
5.2 Combinatorics Module . . . 172
5.3 Number Theory . . . 261
5.4 Basic Cryptography Module . . . 282

5.5 Concrete Mathematics . . . 282
5.6 Numerical evaluation . . . 293
5.7 Numeric Computation . . . 299
5.8 Functions Module . . . 302
5.9 Geometric Algebra Module . . . 398
5.10 Geometry Module . . . 432
5.11 Symbolic Integrals . . . 498
5.12 Numeric Integrals . . . 535
5.13 Logic Module . . . 541
5.14 Matrices . . . 547
5.15 Welcome to mpmath's documentation! . . . 620
5.16 Polynomials Manipulation Module . . . 940
5.17 Printing System . . . 1112
5.18 Plotting Module . . . 1132
5.19 Pyglet Plotting Module . . . 1144
5.20 Assumptions module . . . 1147
5.21 Term rewriting . . . 1162
5.22 Series Expansions . . . 1164
5.23 Sets . . . 1172
5.24 Simplify . . . 1182
5.25 Details on the Hypergeometric Function Expansion Module . . . 1204
5.26 Statistics . . . 1215
5.27 Stats . . . 1222
5.28 ODE . . . 1256
5.29 PDE . . . 1287
5.30 Solvers . . . 1295
5.31 Diophantine . . . 1309
5.32 Tensor Module . . . 1315
5.33 Utilities . . . 1335
5.34 Parsing input . . . 1378
5.35 Calculus . . . 1382
5.36 Physics Module . . . 1382
5.37 Category Theory Module . . . 1563
5.38 Differential Geometry Module . . . 1580
5.39 Contributions to docs . . . 1586

6 Development Tips: Comparisons in Python 1587
6.1 Introduction . . . 1587
6.2 Hashing . . . 1587
6.3 Method Resolution . . . 1588
6.4 General Notes and Caveats . . . 1590
6.5 Script To Verify This Guide . . . 1590

7 Wiki 1595
7.1 FAQ . . . 1595

8 SymPy Papers 1597

9 Planet SymPy 1599

10 SymPy logos 1601

11 Projects using SymPy 1603

12 Blogs, News, Magazines 1605

13 About **1607**
13.1 SymPy Development Team 1607
13.2 Financial and Infrastructure Support 1615
13.3 License 1616

Bibliography **1617**

Python Module Index **1631**

Index **1635**

CHAPTER

ONE

INSTALLATION

The SymPy CAS can be installed on virtually any computer with Python 2.6 or above. SymPy does not require any special Python modules: let us know if you have any problems with SymPy on a standard Python install. The current recommended method of installation is directly from the source files. Alternatively, executables are available for Windows, and some Linux distributions have SymPy packages available.

1.1 Source

SymPy currently recommends that users install directly from the source files. You will first have to download the source files via the archive. Download the latest release (tar.gz) from the downloads site and open it with your operating system's standard decompression utility.

When downloading the archive, make sure to get the correct version (Python 2 or Python 3). If you're not sure which one to use, you probably want the Python 2 version. Note that you can install both if you want.

After the download is complete, you should have a folder called "sympy". From your favorite command line terminal, change directory into that folder and execute the following:

```
$ python setup.py install
```

Alternatively, if you don't want to install the package onto your computer, you may run SymPy with the "isympy" console (which automatically imports SymPy packages and defines common symbols) by executing within the "sympy" folder:

```
$ ./bin/isympy
```

You may now run SymPy statements directly within the Python shell:

```
>>> 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)
>>> diff(x**2/2, x)
x
```

1.2 Git

If you are a developer or like to get the latest updates as they come, be sure to install from git. To download the repository, execute the following from the command line:

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

Then, execute either the *setup.py* or the *bin/isympy* scripts as demonstrated above.

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:

```
$ setupegg.py develop
```

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

1.3 Anaconda

Although SymPy does not have any hard dependencies, many nice features 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 LaTeX printing by running `init_printing()`. An easy way to get all these libraries in addition to SymPy is to install Anaconda, which is a free Python distribution from Continuum Analytics that includes SymPy, Matplotlib, IPython, NumPy, and many more useful packages for scientific computing.

1.4 Other Methods

An installation executable (.exe) is available for Windows users at the downloads site. In addition, various Linux distributions have SymPy available as a package. Others are strongly encouraged to download from source (details above).

1.5 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 5).

1.6 Questions

If you have a question about installation or SymPy in general, feel free to visit the IRC channel at irc.freenode.net, channel #sympy. 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.

CHAPTER

TWO

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 up to 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())
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 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`.

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 16)).

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 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 example 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 $\lim_{x\to 0} \frac{\sin(x)}{x}$.

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

- Solve $x^2 - 2 = 0$.

```
>>> solve(x**2 - 2, x)
```

$$\left[-\sqrt{2}, \sqrt{2}\right]$$

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

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

$$y(t) = C2\cdot e^{-t} + \left(C1 + \frac{t}{2}\right)\cdot e^{t}$$

- Find the eigenvalues of $\left[\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}\right]$.

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

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

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

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

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

- Print $\int_0^{\pi} \cos^2(x)\, dx$ using LaTeX.

```
>>> latex(Integral(cos(x)**2, (x, 0, pi)))
\int_{0}^{\pi} \cos^{2}{\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 even modify the source code and sell 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). Plus, 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 inventing a whole new programming language 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 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.

2.3.1 Printers

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

- str
- repr
- 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.

2.3.2 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 an 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)

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

>>>
```

In any case, this is what will happen:

- In the IPython QTConsole, if LaTeX is installed, it will enable a printer that uses LaTeX.

If 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 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)
⎮
⎮     ╱ ─
⎮    ╲╱ 1
⎮    ─── ─ dx
⎮     ╲╱ 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)
  /
 |
 |     ___
 |    / 1
 |   /  -  dx
 | \/   x
 |
/
>>>
```

To explicitly not use LaTeX, pass `use_latex=False` to `init_printing` or `init_session`. To explicitly not use Unicode, pass `use_unicode=False`.

2.3.3 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)
```

repr

The repr form of an expression is designed to show the exact form of an expression. It will be discussed more in the *Advanced Expression Manipulation* (page 51) section. To get it, use `srepr` [1].

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

The repr form is mostly useful for understanding how an expression is built internally.

ASCII Pretty Printer

The ASCII pretty printer is accessed from `pprint`. If the terminal does not support Unicode, the ASCII printer is used by default. Otherwise, you must pass `use_unicode=False`.

```
>>> pprint(Integral(sqrt(1/x), x), use_unicode=False)
  /
 |
 |     ___
 |    / 1
 |   /  -  dx
 | \/   x
 |
/
```

`pprint` prints the output to the screen. If you want the string form, use `pretty`.

```
>>> pretty(Integral(sqrt(1/x), x), use_unicode=False)
'  /          \n |           \n |     ___    \n |    / 1    \n |   /  -  dx\n | \\/   x    \n |
>>> print(pretty(Integral(sqrt(1/x), x), use_unicode=False))
  /
```

[1] SymPy does not use the Python builtin `repr` function for repr printing, because in Python `str(list)` calls `repr` on the elements of the list, and some SymPy functions return lists (such as `solve`). Since `srepr` is so verbose, it is unlikely that anyone would want it called by default on the output of `solve`.

```
 |
 |      ___
 |     / 1
 |    /  -  dx
 | \/   x
 |
/
```

Unicode Pretty Printer

The Unicode pretty printer is also accessed from `print`. It the terminal supports Unicode, it is used automatically. It `pprint` is not able to detect that the terminal supports unicode, you can pass `use_unicode=True`.

```
>>> pprint(Integral(sqrt(1/x), x), use_unicode=True)
⌠
⎮      ___
⎮     ╱ 1
⎮    ╱  -  dx
⎮  ╲╱   x
⌡
```

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 1121) 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 51) section for some examples of the output of this printer.

2.4 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 `*`.

2.4.1 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
```

2.4.2 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)
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 symbolic 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
```

2.4.3 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
Or(And(Not(x), y), And(Not(y), x))
```

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'>
```

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
```

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
```

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 57) document.

2.4.4 Further Reading

For more discussion on the topics covered in this section, see *Gotchas and Pitfalls* (page 57).

2.5 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 51) section.

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

2.5.1 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 $x^{x^{x^x}}$. 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 51) 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
```

2.5.2 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.

2.5.3 `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)
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067
```

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
```

2.5.4 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]
```

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. Not only accurate for small x.
...     """
...     return x
>>> f = lambdify(x, expr, {"sin":mysin})
>>> f(0.1)
0.1
```

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`, which 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
3·x  - 2·x - 8
--------------
     2
  2·x  - 8·x

>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
   2                2    2            2
x·y  - 2·x·y·z + x·z  + y  - 2·y·z + z
---------------------------------------
                  2
                 x  - 1
>>> cancel(expr)
 2            2
y  - 2·y·z + z
---------------
     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)
       2
(y - z)
--------
 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
   3       2
4·x  + 21·x  + 10·x + 12
------------------------
  4      3      2
 x  + 5·x  + 5·x  + 4·x
>>> apart(expr)
 2·x - 1       1     3
---------- - ----- + -
 2           x + 4   x
x  + x + 1
```

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

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 $(x^a)^b = \left((-1)^2\right)^{1/2} = \sqrt{1} = 1$ and $x^{ab} = (-1)^{2 \cdot 1/2} = (-1)^1 = -1$. However, identity 3 is true at least if x is nonnegative or 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. $(x^a)^b = x^{ab}$	$x \geq 0$ or $b \in \mathbb{Z}$	$\left((-1)^2\right)^{1/2} \neq (-1)^{2 \cdot 1/2}$	$\sqrt{x^2} \neq x$ 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)
```

$$\sqrt{x}\cdot\sqrt{y}$$

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))
```

$$\sqrt{x}\cdot\sqrt{y}$$

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 $\log\left(\frac{x}{y}\right) = \log\left(x \cdot \frac{1}{y}\right) = \log(x) + \log\left(y^{-1}\right) = \log(x) - \log(y)$, 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, $\log\left(e^{x+2\pi i}\right) = \log(e^x) = x \neq x + 2\pi i$).

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))
   ⎛ 2⎞
log⎝z ⎠
>>> 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))
   ⎛ n⎞
log⎝x ⎠
>>> 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)
   ⎛ n⎞
log⎝z ⎠
```

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 303) 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)
Γ(z)
```

The generalized hypergeometric function is `hyper`. `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)
 ┌-  ⎛1, 2 |  ⎞
 ├-  ⎜     | z⎟
2⎹ 1 ⎝ 3   |  ⎠
```

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)
     2
2⋅sin (x)
---------
 sin(2⋅x)
>>> factorial(x).rewrite(gamma)
Γ(x + 1)
```

For some tips on applying more targeted rewriting, see the *Advanced Expression Manipulation* (page 51) section.

expand_func

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

```
>>> expand_func(gamma(x + 3))
x⋅(x + 1)⋅(x + 2)⋅Γ(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 376) for more information).

```
>>> expr = meijerg([[1],[1]], [[1],[]], -z)
>>> expr
╭─╮1, 1 ⎛1  1 |    ⎞
│ ╷     ⎜     | -z ⎟
╰─╯2, 1 ⎝1    |    ⎠
>>> hyperexpand(expr)
 1
 -
 z
e
```

combsimp

To simplify combinatorial expressions, use combsimp.

```
>>> combsimp(factorial(n)/factorial(n - 3))
n·(n - 2)·(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
n + 1
-----
k + 1
```

combsimp also simplifies expressions with gamma.

```
>>> combsimp(gamma(x)*gamma(1 - x))
   π
--------
sin(π·x)
```

2.6.7 Example: Continued Fractions

Let's use SymPy to explore continued fractions. A continued fraction is an expression of the form

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

where $a_0, \ldots, a_n$ are integers, and $a_1, \ldots, a_n$ 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 $[a_0; a_1, \ldots, a_n]$. 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`, ..., `a5`.

```
>>> 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 + --
                     a₄
```

This form is useful for understanding continued fractions, but lets 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
                a2·a₃·a₄ + a2 + a₄
a₀ + ----------------------------------------
     a1·a2·a₃·a₄ + a1·a2 + a1·a₄ + a₃·a₄ + 1
>>> l.append(a0)
>>> frac = 1/(frac - a0)
>>> frac
a1·a2·a₃·a₄ + a1·a2 + a1·a₄ + a₃·a₄ + 1
----------------------------------------
           a2·a₃·a₄ + a2 + a₄
```

Now we repeat this process

```
>>> frac = apart(frac, a1)
>>> frac
         a₃·a₄ + 1
a1 + ------------------
     a2·a₃·a₄ + a2 + a₄
>>> l.append(a1)
>>> frac = 1/(frac - a1)
>>> frac = apart(frac, a2)
>>> frac
         a₄
a2 + ---------
     a₃·a₄ + 1
>>> l.append(a2)
>>> frac = 1/(frac - a2)
>>> frac = apart(frac, a3)
>>> frac
     1
a₃ + --
     a₄
>>> l.append(a3)
>>> frac = 1/(frac - a3)
>>> frac = apart(frac, a4)
>>> frac
a₄
>>> l.append(a4)
>>> list_to_frac(l)
             1
a₀ + -----------------
                1
     a1 + ------------
                  1
          a2 + -------
                    1
               a₃ + --
                    a₄
```

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 $a_0 + \frac{1}{a_1 + \cdots}$ 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⎞
----------⎝e      ⎠
  4   2
∂z  ∂y  ∂x
```

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 44) section).

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 44)).

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))
π
```

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
⎮ 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
 ⎛ x    ⎞      e
log⎝e  + 1⎠ + ------
                2
              x  - 1

>>> integ = Integral(sin(x**2), x)
>>> integ
⎮
⎮     ⎛ 2⎞
⎮ sin⎝x ⎠ dx
⎮
>>> integ.doit()
                        ⎛      ⎞
                        ⎜√2 ⋅x ⎟
3⋅√2 ⋅√π ⋅fresnels⎜-------⎟⋅Γ(3/4)
                        ⎜      ⎟
                        ⎝ √π   ⎠
-------------------------------------
               8⋅Γ(7/4)

>>> integ = Integral(x**y*exp(-x), (x, 0, oo))
>>> integ
∞
⎮
⎮  y  -x
⎮ x ⋅e   dx
⎮
0
>>> integ.doit()
⎧ Γ(y + 1)    for -re(y) < 1
⎪
⎪ ∞
⎪⎮
⎨⎮  y  -x
⎪⎮ x ⋅e   dx    otherwise
⎪
⎪ 0
⎩
```

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

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

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
    cos(x) - 1
lim ----------
x->0    x
>>> expr.doit()
0
```

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

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

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      ⎛ 4⎞
1 + x + -- + O⎝x ⎠
        2
```

The $O\left(x^4\right)$ 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
```

Currently, O only supports orders at 0, so series expansions at points other than 0 are computed by first shifting to 0 and then shifting back.

```
>>> exp(x - 6).series(x, 6)
         2    3    4     5
        x    x    x     x      ⎛ 6⎞
1 + x + -- + -- + -- + --- + O⎝x ⎠
        2    6    24   120
```

This means that if you compute the series expansion at a point other than 0, the result will be shifted to 0. You can easily shift it back with `subs`.

```
>>> exp(x - 6).series(x, 6).removeO().subs(x, x - 6)
           5          4          3          2
    (x - 6)    (x - 6)    (x - 6)    (x - 6)
x + -------- + -------- + -------- + -------- - 5
      120         24         6          2
```

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 18) 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 is 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

```
>>> solve(Eq(x**2, 1), x)
[-1, 1]
>>> solve(Eq(x**2 - 1, 0), x)
[-1, 1]
>>> solve(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 `solve(Eq(expr, 0), x)`, you can just use `solve(expr, x)`.

2.8.2 Solving Equations Algebraically

The main function for solving algebraic equations, as we saw above, is `solve`. The syntax is `solve(equations, variables)`, where, as we saw above, `equations` may be in the form of `Eq` instances or expressions that are assumed to be equal to zero.

When solving a single equation, the output of `solve` is a list of the solutions.

```
>>> solve(x**2 - x, x)
[0, 1]
```

If no solutions are found, an empty list is returned, or `NotImplementedError` is raised.

```
>>> solve(exp(x), x)
[]
```

Note: If `solve` returns `[]` or raises `NotImplementedError`, it doesn't mean that the equation has no solutions. It just means that it couldn't find any. Often this means that the solutions cannot be represented symbolically. For example, the equation $x = \cos(x)$ has a solution, but it cannot be represented symbolically using standard functions.

```
>>> solve(x - cos(x), x)
Traceback (most recent call last):
...
NotImplementedError: multiple generators [x, exp(I*x)]
No algorithms are implemented to solve equation exp(I*x)
```

In fact, `solve` makes *no guarantees whatsoever* about the completeness of the solutions it finds. Much of `solve` is heuristics, which may find some solutions to an equation or system of equations, but not all of them.

`solve` can also solve systems of equations. Pass a list of equations and a list of variables to solve for.

```
>>> solve([x - y + 2, x + y - 3], [x, y])
{x: 1/2, y: 5/2}
>>> solve([x*y - 7, x + y - 6], [x, y])
⎡⎛      ___          ___      ⎞  ⎛      ___          ___      ⎞⎤
⎣⎝- ╲╱ 2  + 3, ╲╱ 2  + 3⎠, ⎝╲╱ 2  + 3, - ╲╱ 2  + 3⎠⎦
```

Note: The type of the output of `solve` when solving systems of equations varies depending on the type of the input. If you want a consistent interface, pass `dict=True`.

```
>>> solve([x - y + 2, x + y - 3], [x, y], dict=True)
[{x: 1/2, y: 5/2}]
>>> solve([x*y - 7, x + y - 6], [x, y], dict=True)
⎡⎧                  ___    ___⎫  ⎧                  ___    ___⎫⎤
⎢⎨x: - \╱ 2  + 3, y: \╱ 2  + 3⎬, ⎨x: \╱ 2  + 3, y: - \╱ 2  + 3⎬⎥
⎣⎩                             ⎭  ⎩                             ⎭⎦
```

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

```
>>> solve(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.

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
                        2
          d            d
f(x) - 2⋅---(f(x)) + ---(f(x)) = sin(x)
          dx           2
                      dx
```

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

```
>>> dsolve(diffeq, f(x))
                    x   cos(x)
f(x) = (C1 + C2·x)·e  + ------
                         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⎤
⎢       ⎥
⎢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
⎡1  0   1⎤
⎢        ⎥
⎢2  -1  3⎥
⎢        ⎥
⎣4  3   2⎦
>>> 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 list of indices of the pivot columns.

```
>>> M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])
>>> M
⎡1   0   1   3 ⎤
⎢              ⎥
⎢2   3   4   7 ⎥
⎢              ⎥
⎣-1  -3  -3  -4⎦
>>> M.rref()
⎛⎡1  0   1    3 ⎤, [0, 1]⎞
⎜⎢               ⎥        ⎟
⎜⎢0  1  2/3  1/3⎥        ⎟
⎜⎢               ⎥        ⎟
⎝⎣0  0   0    0 ⎦        ⎠
```

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

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
⎡1  2   3  0  0⎤
⎢              ⎥
⎣4  10  0  0  1⎦
>>> M.nullspace()
⎡⎡-15⎤, ⎡0⎤, ⎡ 1  ⎤⎤
⎢⎢    ⎥  ⎢ ⎥  ⎢    ⎥⎥
⎢⎢ 6  ⎥  ⎢0⎥  ⎢-1/2⎥⎥
⎢⎢    ⎥  ⎢ ⎥  ⎢    ⎥⎥
```

```
⎡⎡ 1 ⎤  ⎡0⎤  ⎡ 0  ⎤⎤
⎢⎢    ⎥  ⎢ ⎥  ⎢    ⎥⎥
⎢⎢ 0 ⎥  ⎢1⎥  ⎢ 0  ⎥⎥
⎢⎢    ⎥  ⎢ ⎥  ⎢    ⎥⎥
⎣⎣ 0 ⎦  ⎣0⎦  ⎣ 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 44)).

```
>>> 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⎤⎤⎞, ⎛3, 1, ⎡⎡1⎤⎤⎞, ⎛5, 2, ⎡⎡1⎤, ⎡0 ⎤⎤⎞⎤
⎢⎜       ⎢⎢ ⎥⎥⎟  ⎜      ⎢⎢ ⎥⎥⎟  ⎜      ⎢⎢ ⎥  ⎢ ⎥⎥⎟⎥
⎢⎜       ⎢⎢1⎥⎥⎟  ⎜      ⎢⎢1⎥⎥⎟  ⎜      ⎢⎢1⎥  ⎢-1⎥⎥⎟⎥
⎢⎜       ⎢⎢ ⎥⎥⎟  ⎜      ⎢⎢ ⎥⎥⎟  ⎜      ⎢⎢ ⎥  ⎢ ⎥⎥⎟⎥
⎢⎜       ⎢⎢1⎥⎥⎟  ⎜      ⎢⎢1⎥⎥⎟  ⎜      ⎢⎢1⎥  ⎢0 ⎥⎥⎟⎥
⎢⎜       ⎢⎢ ⎥⎥⎟  ⎜      ⎢⎢ ⎥⎥⎟  ⎜      ⎢⎢ ⎥  ⎢ ⎥⎥⎟⎥
⎣⎝       ⎣⎣1⎦⎦⎠  ⎝      ⎣⎣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
(λ - 5) ⋅(λ - 3)⋅(λ + 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 $x^2 + xy$, i.e., `x**2 + 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 = x**2 + x*y
>>> srepr(expr)
"Add(Pow(Symbol('x'), Integer(2)), 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 1131) 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" [2]. 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 `x**2`, this creates a `Pow` object. `Pow` is short for "power".

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

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

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

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` [3].

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

We have seen that `x**2` is represented as `Pow(x, 2)`. 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
```

[2] 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`.

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

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

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

```
>>> expr = x - 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 phenonemon 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 reasons [4].

```
>>> 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.

[4] 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.

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)
```

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
```

CHAPTER

THREE

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 5), 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
>>> 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 67) 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 61) below), numeric evaluation (`N()` is equivalent to *evalf()* (page 293)), the class registry (for things like `C.cos()`, to prevent cyclic imports in some code), 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 `QCOSINE` 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)
>>> 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.
6.20000000000000
>>> 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)  # sympyify 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)
>>> 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 **isympy**, 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)
100.000000000000
>>> 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 aritrary 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 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 can not 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 61) 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 60) 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,
>>> # 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 66), except that it is not

mutable. That means that you can not 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 can not 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 57), 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:

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.
       ...
    """
    expr = Poly.cancel(powsimp(expr))
    return powsimp(together(expr.expand()), combine='exp', deep=True)
```

CHAPTER

FOUR

SYMPY USER'S GUIDE

4.1 Introduction

If you are new to SymPy, start with the *Tutorial* (page 5). 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 81).

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 + 2*x` 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 `2*x`). 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 like:

`cos(x) + 1` is equal to `cos(x).__add__(1)` is equal to `Add(cos(x), Integer(1))`

or

`2/cos(x)` is equal to `cos(x).__rdiv__(2)` is equal to `Mul(Rational(2), Pow(cos(x), Rational(-1)))`.

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_bounded = 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 C.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, it's 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 - -- + --- - ---- + ------ + O\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 - -- + --- - ---- + ------ + O\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 - -- + ---- - ----- + ----- + O\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 - -- + ---- - ----- + ----- + O\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

>>> 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 81) how to fix and improve SymPy's documentation. All contribution is very welcome.

CHAPTER

FIVE

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 this contens:

5.1 SymPy Core

5.1.1 sympify

sympify

`sympy.core.sympify.`**`sympify`**(*a*, *locals=None*, *convert_xor=True*, *strict=False*, *rational=False*)

Converts an arbitrary expression to a type that can be used inside SymPy.

For example, it will convert Python ints into instance of sympy.Rational, 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 them unchanged)
- lists, sets or tuples containing any of the above

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 `bitcout` function is treated as a symbol and the `O` 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("O(x)")
O(x)
>>> sympify("O + 1")
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:

```
>>> ns = {}
>>> exec 'from sympy.core.evalf import bitcount' in ns
>>> sympify(s, locals=ns)
6
```

In order to have the `O` 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["O"] = Symbol("O")  # method 1
>>> exec 'from sympy.abc import O' in ns  # method 2
>>> ns.update(dict(O=Symbol("O")))  # method 3
>>> sympify("O + 1", locals=ns)
O + 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('C & Q', _clash1)
And(C, 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.

```
>>> sympify(True)
True
>>> sympify(True, strict=True)
Traceback (most recent call last):
...
SympifyError: SympifyError: True
```

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
...         raise StopIteration
...     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
...         raise StopIteration
...     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 cache

cacheit

`sympy.core.cache.`**`cacheit`**(*func*)

caching decorator.

important: the result of cached function must be *immutable*

Examples

```
>>> from sympy.core.cache import cacheit
>>> @cacheit
... def f(a,b):
...     return a+b

>>> @cacheit
... def f(a,b):
...     return [a,b] # <-- WRONG, returns mutable object
```

to force cacheit to check returned results mutability and consistency, set environment variable SYMPY_USE_CACHE to 'debug'

5.1.3 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*)

A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.

See docstring of Expr.as_content_primitive

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 Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
set([x, y])

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
set([1, 2])

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
set([1, 2, pi])
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
set([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
set([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))
set([1])

>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
set([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)
set([f(x), sin(y + I*pi)])
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
set([f(x)])

>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
set([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
```

static compare_pretty(**args, **kwargs*)

Is a > b in the sense of ordering in printing?

THIS FUNCTION IS DEPRECATED. Use `default_sort_key` instead.

```
yes ..... return 1
no ...... return -1
equal ... return 0
```

Strategy:

It uses Basic.compare as a fallback, but improves it in many cases, like `x**3`, `x**4`, `O(x**3)` etc. In those simple cases, it just parses the expression and returns the "sane" ordering such as:

```
1 < x < x**2 < x**3 < O(x**4) etc.
```

Examples

```
>>> from sympy.abc import x
>>> from sympy import Basic, Number
>>> Basic._compare_pretty(x, x**2)
-1
>>> Basic._compare_pretty(x**2, x**2)
0
>>> Basic._compare_pretty(x**3, x**2)
1
>>> Basic._compare_pretty(Number(1, 2), Number(1, 3))
1
>>> Basic._compare_pretty(Number(0), Number(-1))
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 symbols method.

Any other method that uses bound variables should implement a 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 xrange(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(**args, **kw_args*)

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 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_Real

Deprecated alias for `is_Float`

is_comparable

Return True if self can be computed to 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
```

is_number

Returns `True` if 'self' contains no free symbols.

See Also:

`is_comparable` (page 90), `sympy.core.expr.is_number`

iter_basic_args()

Iterates arguments of `self`.

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.iter_basic_args()
<...iterator object at 0x...>
>>> list(a.iter_basic_args())
[2, x]
```

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
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

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 subexpression 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 94)** substitution of subexpressions as defined by the objects themselves.

`xreplace` **(page 96)** 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 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(**args, **kw_args*)

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**
  - **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.

- **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 92)** replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

`xreplace` **(page 96)** exact node replacement in expr tree; also capable of using matching rules

Examples

```
>>> from sympy import pi, exp
>>> 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
```

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 92)** replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs **(page 94)** 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, ...

C

sympy.core.basic.**C**

5.1.4 singleton

S

sympy.core.singleton.**S**

5.1.5 expr

5.1.6 Expr

class sympy.core.expr.**Expr**

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 supressed 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]]
>>> (-2*x*y).args_cnc(cset=True)
[set([-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()

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 -> c,e` where x can be any symbolic expression.

as_coeff_mul(**deps*)

Return the tuple (c, args) where self is written as a Mul, `m`.

c should be a Rational multiplied by any terms of the Mul that are independent of deps.

args should be a tuple of all other terms 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 105)** return sum of terms have a given factor

as_coeff_Add **(page 98)** separate the additive constant from an expression

as_coeff_Mul **(page 98)** separate the multiplicative constant from an expression

as_independent **(page 101)** 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
>>> _.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*)

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 no 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)))
```

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.

The returned tuple (i, d) has the following interpretation:

- i will has no variable that appears in deps
- d will be 1 or else have terms that contain variables that are in deps
- if self is an Add then self = i + d
- if self is a Mul then self = i*d
- if self is anything else, either tuple (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

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

See also: .separatevars(), .expand(log=True), .as_two_terms(), .as_coeff_add(), .as_coeff_mul()

as_leading_term(**args, **kw_args*)

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` or None. If `n` is zero then all terms independent of `x` will be returned.

When x is noncommutative, the coeff 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 99)** separate the expression into a coefficient and factor

as_coeff_Add **(page 98)** separate the additive constant from an expression

as_coeff_Mul **(page 98)** separate the multiplicative constant from an expression

as_independent **(page 101)** 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*, *skip_abs=False*, *logx=None*)

as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first. If skip_abs is true, the absolute term is assumed to be zero. (This is necessary because sometimes it cannot be simplified to zero without a lot of work, but is still known to be zero. See log._eval_nseries for an example.) If skip_log is true, log(x) is treated as an independent symbol. (This is needed for the gruntz algorithm.)

could_extract_minus_sign()

Canonical way to choose an element in the set {e, -e} where e is any expression. If the canonical element is e, we have e.could_extract_minus_sign() == True, else e.could_extract_minus_sign() == False.

For any expression, the set `{e.could_extract_minus_sign(), (-e).could_extract_minus_sign()}` must be `{True, False}`.

```
>>> from sympy.abc import x, y
>>> (x-y).could_extract_minus_sign() != (y-x).could_extract_minus_sign()
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(**args*, ***kw_args*)

Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for more information.

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 108), `coeff` (page 105), `as_coefficient` (page 99)

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_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_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 * 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

getO()

Returns the additive O(..) symbol if there is one, else None.

getn()

Returns the order of the expression.

The order is determined either from the O(...) term. If there is no O(...) term, it returns None.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

integrate(**args*, ***kwargs*)

See the integrate function in sympy.integrals

invert(*g*)

See the invert function in sympy.polys

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 112)

References

- http://en.wikipedia.org/wiki/Algebraic_expression

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x')
>>> 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 sin, factor
>>> a = sqrt(sin(x)**2 + 2*sin(x) + 1)/(sin(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 `failing_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()
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. It will be faster than $ifnotself.free_symbols$, however, since is_number will fail as soon as it hits a free symbol.

Examples

```
>>> from sympy import log, Integral
>>> from sympy.abc import x

>>> x.is_number
False
>>> (2*x).is_number
False
>>> (2 + log(2)).is_number
True
>>> (2 + Integral(2, x)).is_number
False
>>> (2 + Integral(2, (x, 1, 2))).is_number
True
```

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)
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)
(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->xlim.

lseries(*x=None*, *x0=0*, *dir='+'*)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for exaxmple, will never terminate. It will just keep printing terms of the sin(x) series:

```
for term in sin(x).lseries(x):
    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, x0 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.

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().

nsimplify(*constants=[ ], tolerance=None, full=False*)
See the nsimplify function in sympy.simplify

powsimp(*deep=False, combine='all'*)
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()
See the radsimp function in sympy.simplify

ratsimp()
See the ratsimp function in sympy.simplify

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
```

refine(*assumption=True*)
See the refine function in sympy.assumptions

removeO()
Removes the additive O(..) 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='+'*)

Series expansion of "self" around `x = x0` yielding either terms of the series one by one (the lazy series given when n=None), else all the terms at once when n != None.

Note: when n != None, if an O() term is returned then the x in the in it and the entire expression represents x - x0, the displacement from x0. (If there is no O() term then the series was exact and x has it's normal meaning.) This is currently necessary since sympy's O() can only represent terms at x0=0. So instead of:

```
cos(x).series(x0=1, n=2) --> (1 - x)*sin(1) + cos(1) + O((x - 1)**2)
```

which graphically looks like this:

```
   |
  .|.         . .
 . | \      .     .
---+----------------------
   |   . .          . .
   |                  x=0
```

the following is returned instead:

```
-x*sin(1) + cos(1) + O(x**2)
```

whose graph is this:

```
   \ |
  . .|        . .
 .   \      .     .
-----+\------------------.
     | . .          . .
     |                  x=0
```

which is identical to `cos(x + 1).series(n=2)`.

Usage: Returns the series expansion of "self" around the point `x = x0` with respect to `x` up to O(x**n) (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)
>>> 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)
>>> [term.next() 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

together(**args, **kwargs*)

See the together function in sympy.polys

trigsimp(**_args_)
See the trigsimp function in sympy.simplify

5.1.7 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, ...

5.1.8 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
```

Wild

class sympy.core.symbol.**Wild**
A Wild symbol matches anything.

Examples

```
>>> from sympy import Wild, WildFunction, cos, pi
>>> from sympy.abc import x
>>> a = Wild('a')
>>> b = Wild('b')
>>> b.match(a)
{a_: b_}
>>> x.match(a)
{a_: x}
>>> pi.match(a)
{a_: pi}
>>> (x**2).match(a)
{a_: x**2}
>>> cos(x).match(a)
{a_: cos(x)}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A_}
```

Dummy

class sympy.core.symbol.**Dummy**

Dummy symbols are each unique, identified by an internal count index:

```
>>> from sympy import Dummy
>>> bool(Dummy("x") == Dummy("x")) == True
False
```

If a name is not supplied then a string value of the count index 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 117) class.

symbols() (page 118) 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(set(['a', 'b', 'c']))
set([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 continguous digits to the left are taken as the nonnegative starting value (or 0 if there are no digit 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('x(:1\,:2)')  # or '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 118) 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 118) with the same arguments and puts the results into the *global* namespace. It's recommended not to use var() (page 120) in library code, where symbols() (page 118) has to be used:

```
>>> 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 120).

5.1.9 numbers

Number

class `sympy.core.numbers.`**`Number`**

Represents any kind of number in sympy.

Floating point numbers are represented by the Float class. Integer numbers (of any size), together with rational numbers (again, there is no limit on their size) are represented by the Rational class.

If you want to represent, for example, `1+sqrt(2)`, then you need to do:

```
Rational(1) + sqrt(Rational(2))
```

as_coeff_Add()
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**
Represents a floating point number. It is capable of representing arbitrary-precision floating-point numbers.

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))
-5.00000000000000
```

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)
3.50000000000000
>>> Float(3)
3.00000000000000
```

Floats can be created from a string representations of Python floats to force ints to Float or to enter high-precision (> 15 significant digits) values:

```
>>> Float('.0010')
0.00100000000000000
>>> Float('1e-3')
0.00100000000000000
>>> Float('1e-3', 3)
0.00100
```

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
```

Rational

class sympy.core.numbers.**Rational**

Represents integers and rational numbers (p/q) of any size.

See Also:

sympify, sympy.simplify.simplify.nsimplify (page 1195)

Examples

```
>>> from sympy import Rational, nsimplify, S, pi
>>> Rational(3)
3
>>> 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")
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
```

as_content_primitive(*radical=False*)

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**

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 Float (page 121)

igcd

sympy.core.numbers.**igcd**(*a, b*)

Computes positive, integer greatest common divisor of two numbers.

The algorithm is based on the well known Euclid's algorithm. To improve speed, igcd() has its own caching mechanism implemented.

ilcm

sympy.core.numbers.**ilcm**(*a, b*)

Computes integer least common multiple of two numbers.

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

E

sympy.core.numbers.**E**

I

sympy.core.numbers.**I**

nan

sympy.core.numbers.**nan**(*cls*)

Not a Number.

This represents the corresponding data type to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`.

NaN serves as a place holder for numeric values that are indeterminate, but not infinite. Most operations on nan, produce another nan. Most indeterminate forms, such as `0/0` or `oo - oo` ‘ produce nan. Three exceptions are ‘‘`0**0`, `1**oo`, and `oo**0`, which all produce `1` (this is consistent with Python’s float).

NaN is a singleton, and can be accessed by `S.NaN`, or can be imported as `nan`.

References

- http://en.wikipedia.org/wiki/NaN

Examples

```
>>> from sympy import nan, S, oo
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
```

oo

sympy.core.numbers.**oo**(*cls*)

pi

sympy.core.numbers.**pi**(*cls*)

zoo

sympy.core.numbers.**zoo**(*cls*)

5.1.10 power

Pow

class sympy.core.power.**Pow**

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*)

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 = floor(y**(1/n)) and a boolean indicating whether the result is exact (that is, whether x**n == y).

```
>>> from sympy import integer_nthroot
>>> integer_nthroot(16,2)
(4, True)
>>> integer_nthroot(26,2)
(5, False)
```

5.1.11 mul

Mul

class sympy.core.mul.**Mul**

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_content_primitive(*radical=False*)

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(**args, **kw_args*)

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. http://code.google.com/p/sympy/issues/detail?id=1497}

    ```
    >>> 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)
    >>> 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. http://code.google.com/p/sympy/issues/detail?id=2629}

    ```
    >>> 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 128) 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. http://code.google.com/p/sympy/issues/detail?id=2607}

  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

5.1.12 add

Add

class sympy.core.add.**Add**

as_coeff_Add()
Efficiently extract the coefficient of a summation.

as_coeff_add(**args, **kw_args*)
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(*a*)
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*)
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 represeting a complex numbers

Examples

```
>>> from sympy import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
```

as_two_terms(**args*, ***kw_args*)

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*y).as_two_terms()
(3, x*y)
```

classmethod class_key()

Nice order of classes

extract_leading_order(**args*, ***kw_args*)

Returns the leading term and it's order.

Examples

```
>>> from sympy.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), O(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, O(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, O(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 129)

primitive()

Return (R, 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 subprocessing 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

5.1.13 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
```

5.1.14 relational

Rel

class sympy.core.relational.**Rel**

A handy wrapper around the Relational class. Rel(a,b, op)

Examples

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x+x**2, '==')
y == x**2 + x
```

Eq

class sympy.core.relational.**Eq**

A handy wrapper around the Relational class. Eq(a,b)

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x, y
>>> Eq(y, x+x**2)
y == x**2 + x
```

Ne

class sympy.core.relational.**Ne**

A handy wrapper around the Relational class. Ne(a,b)

Examples

```
>>> from sympy import Ne
>>> from sympy.abc import x, y
>>> Ne(y, x+x**2)
y != x**2 + x
```

Lt

class sympy.core.relational.**Lt**

A handy wrapper around the Relational class. Lt(a,b)

Examples

```
>>> from sympy import Lt
>>> from sympy.abc import x, y
>>> Lt(y, x+x**2)
y < x**2 + x
```

Le

class sympy.core.relational.**Le**

A handy wrapper around the Relational class. Le(a,b)

Examples

```
>>> from sympy import Le
>>> from sympy.abc import x, y
>>> Le(y, x+x**2)
y <= x**2 + x
```

Gt

class sympy.core.relational.**Gt**

A handy wrapper around the Relational class. Gt(a,b)

Examples

```
>>> from sympy import Gt
>>> from sympy.abc import x, y
>>> Gt(y, x + x**2)
y > x**2 + x
```

Ge

class sympy.core.relational.**Ge**

A handy wrapper around the Relational class. Ge(a,b)

Examples

```
>>> from sympy import Ge
>>> from sympy.abc import x, y
>>> Ge(y, x + x**2)
y >= x**2 + x
```

Equality

class sympy.core.relational.**Equality**

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:

lhs >= rhs

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	(>=)
LessThan	(<=)
StrictGreaterThan	(>)
StrictLessThan	(<)

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	lhs >= rhs
LessThan(lhs, rhs)	lhs <= rhs
StrictGreaterThan(lhs, rhs)	lhs > rhs
StrictLessThan(lhs, rhs)	lhs < 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" when using Python's operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement (1 < x), Python will first recognize the number 1 as a native number, and then that x is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, (x > 1). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement (1 < x) will turn silently into (x > 1).

```
>>> e1 = x >  1
>>> e2 = x >= 1
>>> e3 = x <  1
>>> e4 = x <= 1
>>> e5 = 1 >  x
>>> e6 = 1 >= x
>>> e7 = 1 <  x
>>> e8 = 1 <= x
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
x > 1     x >= 1
x < 1     x <= 1
x < 1     x <= 1
x > 1     x >= 1
```

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, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) >  x
>>> e2 = S(1) >= x
>>> e3 = S(1) <  x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
1 > x     1 >= x
1 < x     1 <= x
1 > x     1 >= x
1 < x     1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

```
>>> e = x < y < z  # silent error!  Where did ``x`` go?
>>> e
y < z
```

Due to an implementation detail or decision of Python [R19] (page 1617), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
And(x < y, y < z)
```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R20] (page 1617)):

```
>>> e = (x < y) < z
>>> type( e )
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge( x, 2 )      # Ge is a convenience wrapper
>>> print e1
x >= 2

>>> rels = Ge( x, 2 ), Gt( x, 2 ), Le( x, 2 ), Lt( x, 2 )
>>> print '%s\n%s\n%s\n%s' % rels
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) 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).

```
>>> e2 = x >= 2
>>> print e2
x >= 2
>>> print "e1: %s,    e2: %s" % (e1, e2)
e1: x >= 2,    e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x > 1
x > 1
x > 1

>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print "%s\n%s\n%s" % rels
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print '%s\n%s\n%s' % rels
```

```
x < 1
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:

lhs >= rhs

In total, there are four *`Than` classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	(>=)
LessThan	(<=)
StrictGreaterThan	(>)
StrictLessThan	(<)

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	lhs >= rhs
LessThan(lhs, rhs)	lhs <= rhs
StrictGreaterThan(lhs, rhs)	lhs > rhs
StrictLessThan(lhs, rhs)	lhs < 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" when using Python's operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement (1 < x), Python will first

recognize the number 1 as a native number, and then that x is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, (x > 1). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement (1 < x) will turn silently into (x > 1).

```
>>> e1 = x >  1
>>> e2 = x >= 1
>>> e3 = x <  1
>>> e4 = x <= 1
>>> e5 = 1 >  x
>>> e6 = 1 >= x
>>> e7 = 1 <  x
>>> e8 = 1 <= x
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
x > 1     x >= 1
x < 1     x <= 1
x < 1     x <= 1
x > 1     x >= 1
```

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, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) >  x
>>> e2 = S(1) >= x
>>> e3 = S(1) <  x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
1 > x     1 >= x
1 < x     1 <= x
1 > x     1 >= x
1 < x     1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

```
>>> e = x < y < z  # silent error!  Where did ``x`` go?
>>> e
y < z
```

Due to an implementation detail or decision of Python [R21] (page 1617), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
And(x < y, y < z)
```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R22] (page 1618)):

```
>>> e = (x < y) < z
>>> type( e )
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge( x, 2 )      # Ge is a convenience wrapper
>>> print e1
x >= 2

>>> rels = Ge( x, 2 ), Gt( x, 2 ), Le( x, 2 ), Lt( x, 2 )
>>> print '%s\n%s\n%s\n%s' % rels
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) 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).

```
>>> e2 = x >= 2
>>> print e2
x >= 2
>>> print "e1: %s,    e2: %s" % (e1, e2)
e1: x >= 2,    e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x > 1
x > 1
x > 1

>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print "%s\n%s\n%s" % rels
```

```
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x < 1
x < 1
x < 1
```

Unequality

class sympy.core.relational.**Unequality**

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:

lhs >= rhs

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	(>=)
LessThan	(<=)
StrictGreaterThan	(>)
StrictLessThan	(<)

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	lhs >= rhs
LessThan(lhs, rhs)	lhs <= rhs
StrictGreaterThan(lhs, rhs)	lhs > rhs
StrictLessThan(lhs, rhs)	lhs < 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" when using Python's operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement (1 < x), Python will first recognize the number 1 as a native number, and then that x is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, (x > 1). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement (1 < x) will turn silently into (x > 1).

```
>>> e1 = x >  1
>>> e2 = x >= 1
>>> e3 = x <  1
>>> e4 = x <= 1
>>> e5 = 1 >  x
>>> e6 = 1 >= x
>>> e7 = 1 <  x
>>> e8 = 1 <= x
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
x > 1     x >= 1
x < 1     x <= 1
x < 1     x <= 1
x > 1     x >= 1
```

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, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) >  x
>>> e2 = S(1) >= x
>>> e3 = S(1) <  x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
1 > x     1 >= x
1 < x     1 <= x
1 > x     1 >= x
1 < x     1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

```
>>> e = x < y < z  # silent error!  Where did ``x`` go?
>>> e
y < z
```

Due to an implementation detail or decision of Python [R23] (page 1618), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
And(x < y, y < z)
```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R24] (page 1619)):

```
>>> e = (x < y) < z
>>> type( e )
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge( x, 2 )      # Ge is a convenience wrapper
>>> print e1
x >= 2

>>> rels = Ge( x, 2 ), Gt( x, 2 ), Le( x, 2 ), Lt( x, 2 )
>>> print '%s\n%s\n%s\n%s' % rels
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) 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).

```
>>> e2 = x >= 2
>>> print e2
x >= 2
>>> print "e1: %s,    e2: %s" % (e1, e2)
e1: x >= 2,    e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x > 1
x > 1
x > 1

>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print "%s\n%s\n%s" % rels
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x < 1
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:

lhs >= rhs

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	(>=)
LessThan	(<=)
StrictGreaterThan	(>)
StrictLessThan	(<)

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	lhs >= rhs
LessThan(lhs, rhs)	lhs <= rhs
StrictGreaterThan(lhs, rhs)	lhs > rhs
StrictLessThan(lhs, rhs)	lhs < 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" when using Python's operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement (1 < x), Python will first recognize the number 1 as a native number, and then that x is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, (x > 1). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement (1 < x) will turn silently into (x > 1).

```
>>> e1 = x >  1
>>> e2 = x >= 1
>>> e3 = x <  1
>>> e4 = x <= 1
>>> e5 = 1 >  x
>>> e6 = 1 >= x
>>> e7 = 1 <  x
>>> e8 = 1 <= x
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
x > 1     x >= 1
x < 1     x <= 1
x < 1     x <= 1
x > 1     x >= 1
```

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, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) >  x
>>> e2 = S(1) >= x
>>> e3 = S(1) <  x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print "%s     %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8)
1 > x     1 >= x
1 < x     1 <= x
1 > x     1 >= x
1 < x     1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

```
>>> e = x < y < z  # silent error!  Where did ``x`` go?
>>> e
y < z
```

Due to an implementation detail or decision of Python [R25] (page 1619), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
And(x < y, y < z)
```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R26] (page 1619)):

```
>>> e = (x < y) < z
>>> type( e )
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge( x, 2 )      # Ge is a convenience wrapper
>>> print e1
x >= 2

>>> rels = Ge( x, 2 ), Gt( x, 2 ), Le( x, 2 ), Lt( x, 2 )
>>> print '%s\n%s\n%s\n%s' % rels
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) 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).

```
>>> e2 = x >= 2
>>> print e2
x >= 2
>>> print "e1: %s,    e2: %s" % (e1, e2)
e1: x >= 2,    e2: x >= 2
```

```
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x > 1
x > 1
x > 1

>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print "%s\n%s\n%s" % rels
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print '%s\n%s\n%s' % rels
x < 1
x < 1
x < 1
```

5.1.15 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 = 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(
```

5.1.16 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.

nargs

The number of arguments that this function takes

variables

The variables used in the internal representation of the function

WildFunction

class sympy.core.function.**WildFunction**

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')
>>> 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)
```

To match functions with more than 1 arguments, set `nargs` to the desired value:

```
>>> F.nargs = 2
>>> f(x, y).match(F)
{F_: f(x, y)}
```

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
>>> 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)
>>> F(c).diff(c)
2
>>> F(c).diff(c)
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 actually 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 actually 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, x, x)
>>> 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),))
```

doit_numerically(*a*, *b*)

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 150)

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, x, x)
>>> diff(f(x), x, 3)
Derivative(f(x), x, x, x)
>>> 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, **kws*)

Base class for function classes. FunctionClass is a subclass of type.

Use Function('<function name>' [, signature]) to create undefined function classes.

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')
>>> 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)
```

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):
...
...     nargs = 1
...
...     @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.

Attributes

nargs	

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 functon is commutative.

classmethod 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".

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)
set([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)
set([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 163), `expand_mul` (page 163), `expand_multinomial` (page 165), `expand_complex` (page 164), `expand_trig` (page 164), `expand_power_base` (page 165), `expand_power_exp` (page 165), `expand_func` (page 164), `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)
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
```

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 + ...)**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.

Example

```
>>> 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)
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)
```

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)
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
```

5.1.17 evalf

PrecisionExhausted

class sympy.core.evalf.**PrecisionExhausted**

N

class sympy.core.evalf.**N**

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
```

5.1.18 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.

```
>>> 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)
```

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 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

5.1.19 compatibility

iterable

sympy.core.compatibility.**iterable**(*i*, *exclude=(<type 'basestring'>, <type 'dict'>)*)

Return a boolean indicating whether `i` is SymPy iterable.

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.

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 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
>>> 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
```

5.1.20 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 171), `sympy.polys.polytools.terms_gcd` (page 955)

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, fraction=False, clear=False)
(x + 2/x)/(2*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 170), `sympy.polys.polytools.terms_gcd` (page 955)

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
```

This only applies when there is a single Add that the coefficient multiplies:

```
>>> factor_terms(x*y/2 + y, clear=True)
y*(x + 2)/2
>>> factor_terms(x*y/2 + y, clear=False) == _
True
```

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 1358), sympy.utilities.iterables.multiset_part (page 1355)

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

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
```

```
{{1, 2}, {3}, {4}, {5}}
>>> _.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)
{{1, 4}, {2}, {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([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 1358), sympy.utilities.iterables.multiset_part (page 1355)

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, ..., 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, ..., 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 superset 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([range(4)])
>>> s = set()
>>> for i in range(20):
...     s.add(a)
...     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-arrangment [R5] (page 1619). 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 198)

References

[R4] (page 1619), [R5] (page 1619), [R6] (page 1619), [R7] (page 1619), [R8] (page 1619), [R9] (page 1619), [R10] (page 1619)

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 ... n-1] [p(0) p(1) p(2) ... p(n-1)]

Since the first line is always 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 range(p.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
```

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 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
Permutation(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
Permutation(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
>>> dict([(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 partion 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)) [R9] (page 1619).

```
>>> 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 184), inversions (page 188), min (page 192), max (page 192)

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()
set([0, 1, 2, 3, 4, 5])
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
set([0, 1, 2, 3, 4, 5])
```

cardinality

Returns the number of all possible permutations.

See Also:

length (page 191), order (page 193), rank (page 194), size (page 196)

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: `~x*~self*x*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.commutes_with(b)
True
>>> b = Permutation([2,3,5,4,1,0])
>>> a.commutes_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 334)

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 181), full_cyclic_form (page 185)

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 182), inversions (page 188), min (page 192), max (page 192)

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')
Permutation(4)(0, 1, 3)
>>> _(sorted("SymPy"))
['S', 'y', 'm', 'P', 'y']
>>> Permutation.from_sequence('SymPy', key=lambda x: x.lower())
Permutation(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 187), `get_precedence_distance` (page 187), `get_adjacency_matrix` (page 186)

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 187), get_precedence_distance (page 187), get_adjacency_distance (page 185)

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 187), get_adjacency_distance (page 185)

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 prededed by job i in both p and p'. This metric is commutative.

See Also:

get_precedence_matrix (page 187), get_adjacency_matrix (page 186), get_adjacency_distance (page 185)

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 187), get_adjacency_matrix (page 186), get_adjacency_distance (page 185)

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 185)

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 -ith element cycling through 0..i.

```
>>> p = Permutation(2)
>>> while p:
...     print 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 184), ascents (page 182), min (page 192), max (page 192)

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 189)

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 193)

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])
>>> 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 189)

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 190)

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 190)

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 countinuing 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 192), `max` (page 192), `suppport`, `cardinality` (page 182), `order` (page 193), `rank` (page 194), `size` (page 196)

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 unmoved 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 192), descents (page 184), ascents (page 182), inversions (page 188)

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 192), descents (page 184), ascents (page 182), inversions (page 188)

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 194), unrank_lex (page 197)

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 194), unrank_nonlex (page 198)

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.

See Also:

rank_trotterjohnson (page 195), unrank_trotterjohnson (page 198)

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 182), length (page 191), rank (page 194), size (page 196)

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 psuedo-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 192), `unrank_lex` (page 197), `cardinality` (page 182), `length` (page 191), `order` (page 193), `size` (page 196)

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 192), unrank_nonlex (page 198)

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 198), next_trotterjohnson (page 193)

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]
```

static 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)^<number of inversions>

See Also:

inversions (page 188)

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 182), length (page 191), order (page 193), rank (page 194)

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([[3, 2], [0, 1]]).size
4
```

support()

Return the elements in permutation, P, for which P[i] != 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 194), next_lex (page 192)

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 192), rank_nonlex (page 194)

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 195), next_trotterjohnson (page 193)

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 that 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)
Cycle(1, 3, 2)
```

The identity cycle, Cycle(), can be used to start a product:

```
>>> Cycle()(1, 2)(2,3)
Cycle(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)
Cycle(1, 5, 4, 10)
>>> Cycle(1, 2)(4)(5)(10)
Cycle(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 177)

Notes

The underlying structure of the Cycle is a dictionary and although the __iter__ method has been redefiend to give the array form of the cycle, the underlying dictionary items are still available with the such methods as items():

```
>>> 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

static generators.**symmetric**(*n*)

Generates the symmetric group of order n, Sn.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import symmetric
>>> list(symmetric(3))
[Permutation(2), Permutation(1, 2), Permutation(2)(0, 1),
 Permutation(0, 1, 2), Permutation(0, 2, 1), Permutation(0, 2)]
```

static generators.**cyclic**(*n*)

Generates the cyclic group of order n, Cn.

See Also:

dihedral (page 200)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import cyclic
>>> list(cyclic(5))
[Permutation(4), Permutation(0, 1, 2, 3, 4), Permutation(0, 2, 4, 1, 3),
 Permutation(0, 3, 1, 4, 2), Permutation(0, 4, 3, 2, 1)]
```

static generators.**alternating**(*n*)

Generates the alternating group of order n, An.

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import alternating
>>> list(alternating(3))
[Permutation(2), Permutation(0, 1, 2), Permutation(0, 2, 1)]
```

static generators.**dihedral**(*n*)

Generates the dihedral group of order 2n, Dn.

The result is given as a subgroup of Sn, except for the special cases n=1 (the group S2) and n=2 (the Klein 4-group) where that's not possible and embeddings in S2 and S4 respectively are given.

See Also:

cyclic (page 200)

Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import dihedral
>>> list(dihedral(3))
[Permutation(2), Permutation(0, 2), Permutation(0, 1, 2),
 Permutation(1, 2), Permutation(0, 2, 1), Permutation(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 228), sympy.combinatorics.permutations.Permutation (page 177)

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_%28permutation_group_theory%29

[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_%28group_theory%29

[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, ..., 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 226), basic_transversals (page 205), basic_orbits (page 204), basic_stabilizers (page 204)

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, ..., b_k)`, this function returns a base `(b_1, b_2, ..., b_{i+1}, b_i, ..., 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 223)

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}^{\left\langle T\right\rangle}|` should be replaced by `|\beta_{i}^{\left\langle T\right\rangle}|`, 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],
[Permutation(0, 1, 2, 3), Permutation(3)(0, 1), Permutation(1, 3, 2),
 Permutation(2, 3), Permutation(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, ..., b_k)` is a base for a group `G`, and `G^{(i)} = G_{b_1, b_2, ..., 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 202), `strong_gens` (page 226), `basic_transversals` (page 205), `basic_stabilizers` (page 204)

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, ..., b_k)` is `G_{b_1, b_2, ..., b_{i-1}}`. For more information, see [1], pp. 87-89.

See Also:

`base` (page 202), `strong_gens` (page 226), `basic_orbits` (page 204), `basic_transversals` (page 205)

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([
    Permutation(3)(0, 1, 2),
    Permutation(1, 2, 3)])
PermutationGroup([
    Permutation(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 226), base (page 202), basic_orbits (page 204), basic_stabilizers (page 204)

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.basic_transversals
[{0: Permutation(3),
  1: Permutation(3)(0, 1, 2),
  2: Permutation(3)(0, 2, 1),
  3: Permutation(0, 3, 1)},
 {1: Permutation(3),
  2: Permutation(1, 2, 3),
  3: Permutation(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 206)

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 \forall 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 226)

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 210)

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 208), `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]*...f[1]*f[0] where n = len(B) and B = G.base. f[i] is one of the permutations in self._basic_orbits[i].

If factor_index==True, returns a tuple [b[0],..,b[n]], where b[i] belongs to self._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 208)

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)
Permutation(7)(2, 4)(3, 5)
```

coset_unrank(*rank, af=False*)

unrank using Schreier-Sims representation

coset_unrank is the inverse operation of coset_rank if 0 <= rank < 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 221)

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())
[Permutation(2), Permutation(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 > \ldots 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, ldots''. :

See Also:

`derived_subgroup` (page 210)

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
>>> 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 209)

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]]
```

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 not true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group()
False
```

But 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([
    Permutation(0, 1)(2, 3),
    Permutation(3),
    Permutation(1, 2, 3),
    Permutation(1, 3, 2),
    Permutation(0, 3, 1),
    Permutation(0, 2, 3),
    Permutation(0, 3)(1, 2),
    Permutation(0, 1, 3),
    Permutation(3)(0, 2, 1),
    Permutation(0, 3, 2),
    Permutation(3)(0, 1, 2),
    Permutation(0, 2)(1, 3)])
>>> _.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]]
```

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
[Permutation(1, 2), Permutation(2)(0, 1)]
```

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])
>>> 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.

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_group()

Return True if the group meets three criteria: identity is present, the inverse of every element is also an element, and the product of any two elements is also an element. If any of the tests fail, False is returned.

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 not a true group:

```
>>> G = tetrahedron.pgroup
>>> G.is_group()
False
```

But the group generated by the permutations in the tetrahedron pgroup is a proper group:

```
>>> H = PermutationGroup(list(G.generate()))
>>> H.is_group()
True
```

The identity permutation is present:

```
>>> H.has(Permutation(G.degree - 1))
True
```

The product of any two elements from the group is also in the group:

```
>>> from sympy import TableForm
>>> g = list(H)
>>> n = len(g)
>>> m = []
>>> for i in g:
...     m.append([g.index(i*H) for H in g])
...
>>> TableForm(m, headings=[range(n), range(n)], wipe_zeros=False)
   | 0  1  2  3  4  5  6  7  8  9  10 11
----------------------------------------
 0 | 11 0  8  10 6  2  7  4  5  3  9  1
 1 | 0  1  2  3  4  5  6  7  8  9  10 11
 2 | 6  2  7  4  5  3  9  1  11 0  8  10
 3 | 5  3  9  1  11 0  8  10 6  2  7  4
 4 | 3  4  0  2  10 6  11 8  9  7  1  5
 5 | 4  5  6  7  8  9  10 11 0  1  2  3
 6 | 10 6  11 8  9  7  1  5  3  4  0  2
 7 | 9  7  1  5  3  4  0  2  10 6  11 8
 8 | 7  8  4  6  2  10 3  0  1  11 5  9
 9 | 8  9  10 11 0  1  2  3  4  5  6  7
10 | 2  10 3  0  1  11 5  9  7  8  4  6
11 | 1  11 5  9  7  8  4  6  2  10 3  0
>>>
The entries in the table give the element in the group corresponding
to the product of a given column element and row element:

>>> g[3]*g[2] == g[9]
True
```

The inverse of every element is also in the group:

```
>>> TableForm([[g.index(~gi) for gi in g]], headings=[[], range(n)],
...    wipe_zeros=False)
0  1 2 3 4  5 6 7 8 9 10 11
--------------------------
11 1 7 3 10 9 6 2 8 5 4  0
```

So we see that g[1] and g[3] are equivalent to their inverse while g[7] == ~g[2].

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 217), `is_solvable` (page 215)

Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
```

```
>>> S.is_nilpotent
False
```

is_normal(*gr*)

Test if G=self is a normal subgroup of gr.

G is normal in gr if for each g2 in G, g1 in gr, g = g1*g2*g1**-1 belongs to G It is sufficient to check this for each g1 in gr.generator and g2 g2 in G.generator

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 219), `random_stab` (page 223)

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 214), derived_series (page 209)

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 > \ldots` 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 `G1` ([1], p.29).

Returns **A list of permutation groups in the order** :

''G = G_0, G_1, G_2, ldots'' :

See Also:

`commutator` (page 206), `derived_series` (page 209)

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 223)

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])
Permutation(0, 1)(2, 3)
>>> G.make_perm(3, [0, 1, 0])
Permutation(0, 2, 3, 1)
>>> G.make_perm([0, 1, 0])
Permutation(0, 2, 3, 1)
```

max_div

Maximum proper divisor of the degree of a permutation group.

See Also:

minimal_block (page 219), _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 ~ on the set `S` such that `a ~ b` implies `g(a) ~ 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), ..., (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, ..., 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 216), `is_primitive` (page 215)

Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.minimal_block([0,5])
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4]
>>> D.minimal_block([0,1])
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

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 `\left\langle S\right\rangle` generated by `S` (for some chosen generating set for `\left\langle S\right\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 206), derived_subgroup (page 210), random_pr (page 223)

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 221)

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)
set([0, 1, 2])
>>> G.orbit([0,4], 'union')
set([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 225)

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)
Permutation(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 `(\beta, g_\beta)`. For a proof of correctness, see [1], p.79

See Also:

orbit (page 220)

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)
[Permutation(5),
 Permutation(0, 1, 2, 3, 4, 5),
 Permutation(0, 5)(1, 4)(2, 3),
 Permutation(0, 2, 4)(1, 3, 5),
 Permutation(5)(0, 4)(1, 3),
 Permutation(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()
[set([0, 2, 3, 4, 6]), set([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 209)

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())
[Permutation(2), Permutation(2)(0, 1)]

>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> 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,\ldots, p_k\}, the pointwise stabilizer of p_1, p_2, \ldots, p_k is defined as G_{p_1,\ldots, p_k} = \{g\in G | g(p_i) = p_i \forall i\in\{1, 2,\ldots,k\}\} ([1],p20). It is a subgroup of ‘‘G.

See Also:

stabilizer (page 226), schreier_sims_incremental (page 223)

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
```

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 223), `orbit_rep` (page 220)

schreier_sims()

Schreier-Sims algorithm.

It computes the generators of the chain of stabilizers G > G_{b_1} > .. > G_{b1,..,b_r} > 1 in which G_{b_1,..,b_i} stabilizes b_1,..,b_i, and the corresponding `s` cosets. An element of the group can be written as the product h_1*..*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: Permutation(2)(0, 1), 1: Permutation(2), 2: Permutation(1, 2)},
 {0: Permutation(2), 2: Permutation(0, 2)}]
```

schreier_sims_incremental(*base=None, gens=None*)

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`.

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 223), `schreier_sims_random` (page 224)

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^{-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 223)

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 220)

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 `\alpha` is the group `G_\alpha = \{g \in G | g(\alpha) = \alpha\}`. For a proof of correctness, see [1], p.79.

See Also:

`orbit` (page 220)

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.stabilizer(5)
PermutationGroup([
    Permutation(5)(0, 4)(1, 3),
    Permutation(5)])
```

strong_gens

Return a strong generating set from the Schreier-Sims algorithm.

A generating set `S = \{g_1, g_2, ..., 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, ..., 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, ..., 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 202), `basic_transversals` (page 205), `basic_orbits` (page 204), `basic_stabilizers` (page 204)

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> D.strong_gens
[Permutation(0, 1, 2, 3), Permutation(0, 3)(1, 2), Permutation(1, 3)]
>>> D.base
[0, 1]
```

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 lenghty 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 mentiones that this can be done by using `.baseswap(...)`, however the current imlementation 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)
>>> G.is_subgroup(AlternatingGroup(7))
True
```

```
>>> _verify_bsgs(G, base, G.generators)
True
```

transitivity_degree

Compute the degree of transitivity of the group.

A permutation group `G` acting on `\Omega = \{0, 1, ..., n-1\}` is `k`-fold transitive, if, for any `k` points `(a_1, a_2, ..., a_k)\in\Omega` and any `k` points `(b_1, b_2, ..., b_k)\in\Omega` there exists `g\in G` such that `g(a_1)=b_1, g(a_2)=b_2, ..., 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 216), `orbit` (page 220)

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 229), `cyclic_form` (page 229)

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 228), `cyclic_form` (page 229)

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 229), `array_form` (page 228)

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 228), `cyclic_form` (page 229)

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

[R11] (page 1619)

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 232), rank (page 233), prev (page 232), size (page 233)

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 232), rank (page 233), next (page 232), size (page 233)

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 233), next (page 232), prev (page 232), size (page 233)

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 continuuing until only two verices remain. The Prufer sequence is the list of recorded nodes.

See Also:

to_prufer (page 234)

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 232), next (page 232), prev (page 232), size (page 233)

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 232), rank (page 233), next (page 232), prev (page 232)

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 233)** 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 234)** returns tree representation of a Prufer object.

References

- http://hamberg.no/erlend/2010/11/06/prufer-sequence/

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 234)

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 239)

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 239), superset (page 240), size (page 239), superset_size (page 240)

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 236), prev_binary (page 237)

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 237), prev_gray (page 238)

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 237), iterate_binary (page 236)

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 236), prev_gray (page 238)

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 238)

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 236), iterate_binary (page 236)

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 236), next_gray (page 237)

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 237)

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 236), unrank_binary (page 240)

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 236), unrank_gray (page 241)

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 239), superset (page 240), superset_size (page 240), cardinality (page 235)

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 240), size (page 239), superset_size (page 240), cardinality (page 235)

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 235)

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 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 239), size (page 239), superset_size (page 240), cardinality (page 235)

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 239), superset (page 240), size (page 239), cardinality (page 235)

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 236), rank_binary (page 238)

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 236), rank_gray (page 238)

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
[]
```

static subsets.**ksubsets**(*superset*, *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', '1
```

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 243)

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://www-stat.stanford.edu/~susan/courses/s208/node12.html

See Also:

`unrank` (page 244)

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 242)

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 243)

Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

static graycode.**random_bitstring**(*n*)

Generates a random bitlist of length n.

Examples

```
>>> from sympy.combinatorics.graycode import random_bitstring
>>> random_bitstring(3)
100
```

static graycode.**gray_to_bin**(*bin_list*)

Convert from Gray coding to binary coding.

We assume big endian encoding.

See Also:

bin_to_gray (page 245)

Examples

```
>>> from sympy.combinatorics.graycode import gray_to_bin
>>> gray_to_bin('100')
'111'
```

static graycode.**bin_to_gray**(*bin_list*)

Convert from binary coding to gray coding.

We assume big endian encoding.

See Also:

gray_to_bin (page 244)

Examples

```
>>> from sympy.combinatorics.graycode import bin_to_gray
>>> bin_to_gray('111')
'100'
```

static graycode.**get_subset_from_bitstring**(*super_set, bitstring*)

Gets the subset defined by the bitstring.

See Also:

graycode_subsets (page 245)

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']
```

static graycode.**graycode_subsets**(*gray_code_set*)

Generates the subsets as enumerated by a Gray code.

See Also:

get_subset_from_bitstring (page 245)

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'],
```

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 ... 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 246), DihedralGroup (page 246), AlternatingGroup (page 247)

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()
False
>>> 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 ... n-1) (in cycle notation). After the group is generated, some of its basic properties are set.

See Also:

SymmetricGroup (page 246), DihedralGroup (page 246), AlternatingGroup (page 247)

Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(6)
>>> G.is_group()
False
>>> G.order()
6
>>> 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 ... n-1)` (a rotation of the n-gon) and `b = (0 n-1)(1 n-2)...` (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` so they indeed generate D_n (See [1]). After the group is generated, some of its basic properties are set.

See Also:

SymmetricGroup (page 246), CyclicGroup (page 246), AlternatingGroup (page 247)

References

[1] http://en.wikipedia.org/wiki/Dihedral_group

Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group()
False
>>> 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 ... n-1)` for n odd and `(0 1 2), (1 2 ... 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 246), CyclicGroup (page 246), DihedralGroup (page 246)

References

[1] Armstrong, M. “Groups and Symmetry”

Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group()
False
>>> 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. [1] http://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups

See Also:

`DirectProduct`

Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> AbelianGroup(3, 4)
PermutationGroup([
        Permutation(6)(0, 1, 2),
        Permutation(3, 4, 5, 6)])
>>> _.is_group()
False
```

Utilities

sympy.combinatorics.util.**_base_ordering**(*base*, *degree*)

Order $\{0, 1, ..., n-1\}$ 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. :

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] :

Notes

This is used in backtrack searches, when we define a relation $<<$ on the underlying set for a permutation group of degree n, $\{0, 1, ..., n-1\}$, so that if $(b_1, b_2, ..., b_k)$ is a base we have $b_i << b_j$ whenever $i < j$ and $b_i << a$ for all $i \in \{1, 2, ..., k\}$ 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"

`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 212)

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 $(b_1, b_2, ..., b_k)$, the basic stabilizers are defined as $G^{(i)} = G_{b_1,...,b_{i-1}}$ for $i \in \{1, 2, ..., k\}$.

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 253), `_orbits_transversals_from_bsgs` (page 251), `_handle_precomputed_bsgs` (page 250)

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
[Permutation(0, 1, 2), Permutation(0, 2), Permutation(1, 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[Permutation(0, 1, 2), Permutation(0, 2), Permutation(1, 2)],
 [Permutation(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 251), 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: Permutation(2), 1: Permutation(0, 1, 2), 2: Permutation(0, 2)},
{1: Permutation(2), 2: Permutation(1, 2)}],
[[0, 1, 2], [1, 2]], [[Permutation(0, 1, 2),
                       Permutation(0, 2),
                       Permutation(1, 2)],
                      [Permutation(1, 2)]])
```

sympy.combinatorics.util.**_orbits_transversals_from_bsgs**(*base, strong_gens_distr, transversals_only=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 swithing between returning only the :

transversals/ both orbits and transversals :

See Also:

`_distribute_gens_by_base` (page 249), `_handle_precomputed_bsgs` (page 250)

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: Permutation(2), 1: Permutation(0, 1, 2), 2: Permutation(0, 2, 1)},
{1: Permutation(2), 2: Permutation(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()
>>> len(strong_gens)
26
>>> 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 infromation 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 223), sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims_random (page 224)

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)
(Permutation(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 249)

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
[Permutation(0, 1, 2), Permutation(2)(0, 1), Permutation(1, 2)]
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _strong_gens_from_distr(strong_gens_distr)
[Permutation(0, 1, 2), Permutation(2)(0, 1), Permutation(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])
>>> 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 223)

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 254), sympy.combinatorics.perm_groups.PermutationGroup.cent (page 206), _cmp_perm_lists (page 254)

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 do 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

$T = A_{d0d1} * B^{d0}{}_{d2} * B^{d2d1}$

$ord = [d0, -d0, d1, -d1, d2, -d2]$ 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_direc
>>> 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])
>>> canonicalize(g, range(6), 0, t0, t1)
0
```

same as above, but with B_{ab} anticommuting

$T_c = -A^{d0d1} * B_{d0}{}^{d2} * B_{d1d2}$

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)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base1, gens1, base1, gens1)
>>> 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 [d0, -d0, d1, -d1, ...].

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 $t->T(ind[(g*s)[0]], ..., ind[(g*s)[n-1]])$

A dummy symmetry transformation acts on ind $t->T(ind[(d*g)[0]], ..., ind[(d*g)[n-1]])$

Being interested only in the transformations of the tensor under these symmetries, one can represent the tensor by g, which transforms as

$g->d*g*s$, 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 $-(2, 4)$;

$= T_{d3d1}{}^{d3d1}{}_{d2}{}^{d2}$ 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 $rep = min(D * g * S) = min([d * g * s for d in D for s in S])$

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

$S -> S_{b0} -> S_{b0,b1} -> ...$ and $D -> D_{p0} -> D_{p0,p1} -> ...$

where $[b0, b1, ...] = range(b)$ 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.

$dbase = [p0, p1, ...]$ 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 $0 <= j < i$ starting from $s_0 = id, d_0 = id, h_0 = g$.

The equations $h_0[0] = p_0, h_1[1] = p_1, ...$ are solved in this order, choosing each time the lowest possible value of p_i

For $j < i$ $d_i * g * s_i * S_{b_0,...,b_{i-1}} * b_j = D_{p_0,...,p_{i-1}} * p_j$ so that for dx in $D_{p_0,...,p_{i-1}}$ and sx in $S_{base[0],...,base[i-1]}$ 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 $s_i * sx * b_j = J, dx * d_i * g * J = p_j$ $sx * b_j = s_i ** -1 * J; sx = trace(s_i ** -1, S_{b_0,...,b_{i-1}})$ $dx ** -1 * p_j = d_i * g * J; dx = trace(d_i * g * J, D_{p_0,...,p_{i-1}})$

$s_{i+1} = s_i * trace(s_i ** -1 * J, S_{b_0,...,b_{i-1}})$ $d_{i+1} = trace(d_i * g * J, D_{p_0,...,p_{i-1}}) ** -1 * d_i$ $h_{i+1} * b_i = d_{i+1} * g * s_{i+1} * b_i = p_i$

$h_n * b_j = p_j$ 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([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([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], [Permutation(4)(0, 1), Permutation(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], [Permutation(4)(1, 2)])
```

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.

```
>>> from sympy import sieve
>>> from array import array # this line and next for doctest only
>>> sieve._list = array('l', [2, 3, 5, 7, 11, 13])

>>> 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 <= n (a real number).

Examples

```
>>> from sympy import sieve
>>> from array import array # this line and next for doctest only
>>> sieve._list = array('l', [2, 3, 5, 7, 11, 13])

>>> sieve.extend(30)
>>> sieve[10] == 29
True
```

extend_to_no(i)

Extend to include the ith 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
>>> from array import array # this line and next for doctest only
>>> sieve._list = array('l', [2, 3, 5, 7, 11, 13])
```

```
>>> sieve.extend_to_no(9)
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

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 n error will be raised.

Examples

```
>>> from sympy import sieve
>>> sieve.search(25)
(9, 10)
>>> sieve.search(23)
(9, 9)
```

5.3.2 Ntheory Functions Reference

sympy.ntheory.generate.**prime**(*nth*)

Return the nth prime, with the primes indexed as prime(1) = 2, prime(2) = 3, etc.... The nth prime is approximately n*log(n) and can never be larger than 2**n.

See Also:

sympy.ntheory.primetest.isprime **(page 279)** Test if n is prime

primerange **(page 264)** Generate all primes in a given range

primepi **(page 263)** Return the number of primes less than or equal to n

References

- http://primes.utm.edu/glossary/xpage/BertrandsPostulate.html

Examples

```
>>> from sympy import prime
>>> prime(10)
29
>>> prime(1)
2
```

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.

See Also:

sympy.ntheory.primetest.isprime **(page 279)** Test if n is prime

primerange **(page 264)** Generate all primes in a given range

prime **(page 262)** 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 263)** Return the largest prime smaller than n

primerange **(page 264)** Generate all primes in a given range

Notes

Potential primes are located at 6*j +/- 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 263)** Return the ith prime greater than n

primerange **(page 264)** Generates all primes in a given range

Notes

Potential primes are located at 6*j +/- 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 modifiy the sieve.

See Also:

`nextprime` **(page 263)** Return the ith prime greater than n

`prevprime` **(page 263)** Return the largest prime smaller than n

`randprime` **(page 265)** Returns a random prime in a given range

`primorial` **(page 265)** Returns the product of primes based on condition

`Sieve.primerange` **(page 262)** return range from already computed primes or extend the sieve to contain the requested range.

Notes

Some famous conjectures about the occurence 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 ... 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 264)** 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 264)** 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:

```
>>> cycle_length(func, 4).next()
(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):

```
>>> cycle_length(func, 4, nmax = 4).next()
(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.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 272), smoothness_p (page 267)

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 + n) 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 +/- 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 272), smoothness (page 266)

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 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' % cycle_length(F, s).next()
...
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
>>> cycle_length(lambda x: (x**2+12)%17, 2).next()
(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 kth and 2*kth 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 lcm(1..B) = reduce(ilcm, 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 mumber 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])
set([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 != 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 divisisor 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*)

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
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, 2029167820761624560228770248 59L: 1, 7: 1}
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}
```

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, **dict(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 266), `smoothness_p` (page 267), `divisors` (page 274)

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 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.

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 274)

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 274), `factorint` (page 272), `divisor_count` (page 275)

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 272), `divisors` (page 274), `totient` (page 275)

References

- http://www.mayer.dial.pipex.com/maths/formulae.htm

```
>>> from sympy import divisor_count
>>> divisor_count(6)
4
```

sympy.ntheory.factor_.**totient**()

Calculate the Euler totient function phi(n)

```
>>> from sympy.ntheory import totient
>>> totient(1)
1
>>> totient(25)
20
```

See Also:

`divisor_count` (page 275)

Attributes

nargs	

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 mod 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 277)

`sympy.polys.galoistools.gf_crt` **(page 1079)** 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 `ai` when it is divided by `mi` where the `ai` and `mi` are given as pairs to this function: ((a1, m1), (a2, m2), ...). If there is no solution, return None. Otherwise return `n` and its modulus.

The `mi` 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 275)** 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 $(k1, k2) : C_kn$ where C_kn are binomial coefficients and $n = k1 + k2$. 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 278), multinomial_coefficients (page 278)

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 277), multinomial_coefficients (page 278)

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 `{(k1,k2,..,km) : C_kn}` where `C_kn` are multinomial coefficients such that `n=k1+k2+..+km`.

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:

$$\binom{n}{k_1, \ldots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^{m} \binom{n}{k_1 + 1, \ldots, k_i - 1, \ldots}$$

Code contributed to Sage by Yann Laigle-Chapuy, copied with permission of the author.

See Also:

binomial_coefficients_list (page 278), binomial_coefficients (page 277)

sympy.ntheory.multinomial.**multinomial_coefficients_iterator**(*m*, *n*, *_tuple=<type '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)
>>> it.next()
((3, 0, 0, 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, described e.g. at http://mathworld.wolfram.com/PartitionFunctionP.html

The correctness of this implementation has been tested for 10**n up to n = 8.

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 < 10**16 the answer is accurate; greater n values have a small probability of actually being pseudoprimes.

Negative primes (e.g. -2) are not considered prime.

The function first looks for trivial factors, and if none is found, performs a safe Miller-Rabin strong pseudoprime test with bases that are known to prove a number prime. Finally, a general Miller-Rabin test is done with the first k bases which will report a pseudoprime as a prime with an error of about 4**-k. The current value of k is 46 so the error is about 2 x 10**-28.

See Also:

sympy.ntheory.generate.primerange **(page 264)** Generates all primes in a given range

sympy.ntheory.generate.primepi **(page 263)** Return the number of primes less than or equal to n

sympy.ntheory.generate.prime **(page 262)** Return the nth prime

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 totient(p) is the smallest positive number s.t.

a**totient(p) cong 1 mod(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.**is_quad_residue**(*a*, *p*)

Returns True if a (mod p) is in the set of squares mod p, i.e a % p in set([i**2 % p for i 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
>>> list(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 281), jacobi_symbol (page 281)

sympy.ntheory.residue_ntheory.**legendre_symbol**(*a*, *p*)

Returns **1. 0 if a is multiple of p** :

2. 1 if a is a quadratic residue of p :

3. -1 otherwise :

p should be an odd prime by definition :

See Also:

is_quad_residue (page 281), jacobi_symbol (page 281)

Examples

```
>>> from sympy.ntheory import legendre_symbol
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, 1, -1, -1]
>>> list(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
```

sympy.ntheory.residue_ntheory.**jacobi_symbol**(*m*, *n*)

Returns the product of the legendre_symbol(m, p) for all the prime factors, p, of n.

Returns **1. 0 if m cong 0 mod(n)** :

2. 1 if x2 cong m mod(n) has a solution** :

3. -1 otherwise :

See Also:

is_quad_residue (page 281), legendre_symbol (page 281)

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
```

5.4 Basic Cryptography Module

Included in this module are both block ciphers and stream ciphers.

- Shift cipher
- Affine cipher
- Bifid ciphers
- Vigenere's cipher
- substitution ciphers
- Hill's cipher
- RSA
- Kid RSA
- linear feedback shift registers (a stream cipher)
- ElGamal encryption

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 is 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`, `Product`, `product`

References

[R12] (page 1620), [R13] (page 1620), [R14] (page 1620)

Examples

```
>>> from sympy.abc import i, k, m, n, x
>>> from sympy import Sum, factorial, oo
>>> Sum(k,(k,1,m))
Sum(k, (k, 1, m))
>>> 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)
```

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:

$$\sum_{m \le i < n} f(i)$$

has *the obvious meaning* for $m < n$, namely:

$$\sum_{m \le i < n} f(i) = f(m) + f(m+1) + \ldots + f(n-2) + f(n-1)$$

with the upper limit value $f(n)$ excluded. The sum over an empty set is zero if and only if $m = n$:

$$\sum_{m \le i < n} f(i) = 0 \quad \text{for} \quad m = n$$

Finally, for all other sums over empty sets we assume the following definition:

$$\sum_{m \le i < n} f(i) = -\sum_{n \le i < m} f(i) \quad \text{for} \quad m > n$$

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:

$$\sum_{m \le i < n} f(i) = \sum_{i=m}^{n-1} f(i)$$

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.

is_number

Return True if the Sum will result in a number, else False.

Sums are a special case since they contain symbols that can be replaced with numbers. Whether the integral can be done or not is another issue. But answering whether the final result is a number is not difficult.

Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y
>>> Sum(x, (y, 1, x)).is_number
False
>>> Sum(1, (y, 1, x)).is_number
False
>>> Sum(0, (y, 1, x)).is_number
True
>>> Sum(x, (y, 1, 2)).is_number
False
>>> Sum(x, (y, 1, 1)).is_number
False
>>> Sum(x, (x, 1, 2)).is_number
True
>>> Sum(x*y, (x, 1, 2), (y, 1, 3)).is_number
True
```

is_zero

A Sum is only zero if its function is zero or if all terms cancel out. This only answers whether the summand zero.

variables

Return a list of the summation variables

```
>>> from sympy import Sum
>>> from sympy.abc import x, i
>>> Sum(x**i, (i, 1, 3)).variables
[i]
```

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`

References

[R15] (page 1620), [R16] (page 1620), [R17] (page 1620)

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()
nan
```

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)*
>>> Pe = simplify(Pe)
>>> 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:

$$\prod_{m \leq i < n} f(i) = f(m)f(m+1) \cdot \ldots \cdot f(n-2)f(n-1)$$

with the upper limit value $f(n)$ excluded. The product over an empty set is one if and only if $m = n$:

$$\prod_{m \leq i < n} f(i) = 1 \quad \text{for} \quad m = n$$

Finally, for all other products over empty sets we assume the following definition:

$$\prod_{m \leq i < n} f(i) = \frac{1}{\prod_{n \leq i < m} f(i)} \quad \text{for} \quad m > n$$

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.

free_symbols

This method returns the symbols that will affect the value of the Product when evaluated. This is useful if one is trying to determine whether a product depends on a certain symbol or not.

```
>>> from sympy import Product
>>> from sympy.abc import x, y
>>> Product(x, (x, y, 1)).free_symbols
set([y])
```

is_number

Return True if the Product will result in a number, else False.

Examples

```
>>> from sympy import log, Product
>>> from sympy.abc import x, y, z
>>> log(2).is_number
True
>>> Product(x, (x, 1, 2)).is_number
True
>>> Product(y, (x, 1, 2)).is_number
False
>>> Product(1, (x, y, z)).is_number
True
>>> Product(2, (x, y, z)).is_number
False
```

is_zero

A Product is zero only if its term is zero.

variables

Return a list of the product variables

```
>>> from sympy import Product
>>> from sympy.abc import x, i
>>> Product(x**i, (i, 1, 3)).variables
[i]
```

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.,

```
                                b
                              ____,
                              \
summation(f, (i, a, b)) =      )    f
                              /___,
                              i = a
```

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, Product, product

sympy.concrete.products.**product**(**args, **kwargs*)

Compute the product.

The notation for symbols is similiar 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.,

```
                                b
                              _____
product(f(n), (i, a, b)) = |    | f(n)
                            |    |
                            i = a
```

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

[R18] (page 1620)

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)
0.28902548222223624241 - 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 pi:

```
>>> str(N(pi, 10**6))[-50:]
'95678796130331164628399634646042209010610577945815'
```

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.

5.6.2 Floating-point numbers

Floating-point numbers in SymPy are instances of the class `Float`. A `Float` can be created with a custom precision as second argument:

```
>>> Float(0.1)
0.100000000000000
>>> Float(0.1, 10)
0.1000000000
>>> Float(0.125, 30)
0.125000000000000000000000000000
>>> Float(0.1, 30)
0.100000000000000005551115123126
```

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 .125 = 1/4) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:

```
>>> Float('0.1', 30)
0.100000000000000000000000000000
>>> Float(Rational(1, 10), 30)
0.100000000000000000000000000000
>>> Rational(1, 10).evalf(30)
0.100000000000000000000000000000
```

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.

```
>>> Float(0.1, 3)*Float(3.1415, 5)
0.31417
```

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.

`N` and `evalf` can be used to change the precision of existing floating-point numbers:

```
>>> N(3.5)
3.50000000000000
>>> N(3.5, 5)
3.5000
>>> N(3.5, 30)
3.50000000000000000000000000000
```

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 + 434665576869374564356885276750406258025646605173717804024817290895365

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)
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)
3.00000000000000
```

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 discontinuities. 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)
>>> intgrl = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgrl
 oo
  /
 |
 |  sin(x)
 |  ------ dx
 |     2
 |    x
 |
/
1
>>> N(intgrl, 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
>>> 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 pi 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)
3.14159265358979323846264338327950288419716939937510582097494459230781640628620 8
9986280348253421170679821480865132823066470938446095505822317253594081284811174 5
0284102701938521105559644622948954930381964428810975665933446128475648233786783 1
...
```

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. pi) 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
>>> nsimplify(2.0**(1/3.), tolerance=0.001)
635
```

```
---
504
>>> nsimplify(2.0**(1/3.), tolerance=0.001, full=True)
3 ___
\/ 2
```

Here are several more advanced examples:

```
>>> nsimplify(Float('0.13019886662998677236912797033 7',30), [pi, E])
    1
----------
5*pi
---- + 2*E
 7
>>> nsimplify(cos(atan('1/3')))
    ____
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)))
           __________
          /    ___
1        /  \/ 5    1
- - I*  /   ----- + -
2     \/      10    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])
   ___
\/ 2 *pi
```

5.7 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`.

5.7.1 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
>>> 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.

5.7.2 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.

5.7.3 uFuncify

While NumPy operations are very efficient for vectorized data 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

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

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

```
>>> 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 blogpost on this topic http://ojensen.wordpress.com/2010/08/10/fast-ufunc-ish-hydrogen-solutions/

5.7.4 Theano

SymPy has a strong connection with [Theano](http://deeplearning.net/software/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.

5.7.5 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
subs/evalf	50us	Simple	None
lambdify	1us	Scalar functions	math
lambdify-numpy	10ns	Vector functions	numpy
ufuncify	10ns	Complex vector expressions	f2py, Cython
Theano	10ns	Many outputs, CSE, GPUs	Theano

5.8 Functions Module

All functions support the methods documented below, inherited from `sympy.core.function.Function` (page 154).

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)
```

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):
...
...     nargs = 1
...
...     @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.

Attributes

nargs	

`Function.`**`as_base_exp`**`()`
Returns the method as the 2-tuple (base, exponent).

`Function.`**`fdiff`**`(`*argindex=1*`)`
Returns the first derivative of the function.

`Function.`**`is_commutative`**
Returns whether the functon is commutative.

classmethod `Function.`**`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".

5.8.1 Contents

Elementary

This module implements elementary functions, as well as functions like `Abs`, `Max`, etc.

Abs

Returns the absolute value of the argument.

Examples:

```
>>> from sympy.functions import Abs
>>> Abs(-1)
1
```

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, conjugate

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)
```

acos

class sympy.functions.elementary.trigonometric.**acos**

See Also:

asin, atan, cos

Notes

- acos(x) will evaluate automatically in the cases oo, -oo, 0, 1, -1

Examples

```
>>> from sympy import acos, oo, pi
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I
```

inverse(*argindex=1*)
Returns the inverse of this function.

acosh

class sympy.functions.elementary.hyperbolic.**acosh**
The inverse hyperbolic cosine function.

- acosh(x) -> Returns the inverse hyperbolic cosine of x

See Also:

asinh, atanh, cosh

inverse(*argindex=1*)
Returns the inverse of this function.

acot

class sympy.functions.elementary.trigonometric.**acot**

inverse(*argindex=1*)
Returns the inverse of this function.

acoth

class sympy.functions.elementary.hyperbolic.**acoth**
The inverse hyperbolic cotangent function.

- acoth(x) -> Returns the inverse hyperbolic cotangent of x

inverse(*argindex=1*)
Returns the inverse of this function.

arg

Returns the argument (in radians) of a complex number. For a real 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
```

class `sympy.functions.elementary.complexes.`**`arg`**

Returns the argument (in radians) of a complex number

asin

class `sympy.functions.elementary.trigonometric.`**`asin`**

See Also:

`acos`, `atan`, `sin`

Notes

- asin(x) will evaluate automatically in the cases oo, -oo, 0, 1, -1

Examples

```
>>> from sympy import asin, oo, pi
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2
```

`inverse`(*argindex=1*)

Returns the inverse of this function.

asinh

class `sympy.functions.elementary.hyperbolic.`**`asinh`**

The inverse hyperbolic sine function.

- asinh(x) -> Returns the inverse hyperbolic sine of x

See Also:

`acosh`, `atanh`, `sinh`

`inverse`(*argindex=1*)

Returns the inverse of this function.

atan

class sympy.functions.elementary.trigonometric.**atan**

See Also:

acos, asin, tan

Notes

- atan(x) will evaluate automatically in the cases oo, -oo, 0, 1, -1

Examples

```
>>> from sympy import atan, oo, pi
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2
```

inverse(*argindex=1*)
Returns the inverse of this function.

atan2

This function is like atan, but considers the sign of both arguments in order to correctly determine the quadrant of its result.

class sympy.functions.elementary.trigonometric.**atan2**
atan2(y,x) -> Returns atan(y/x) taking two arguments y and x. Signs of both y and x are considered to determine the appropriate quadrant of atan(y/x). The range is $(-\pi, \pi]$.

atanh

class sympy.functions.elementary.hyperbolic.**atanh**
The inverse hyperbolic tangent function.

- atanh(x) -> Returns the inverse hyperbolic tangent of x

See Also:

asinh, acosh, tanh

inverse(*argindex=1*)
Returns the inverse of this function.

ceiling

class sympy.functions.elementary.integers.**ceiling**
Ceiling is a univariate function which returns the smallest integer value not less than its argument. Ceiling function is generalized in this implementation to complex numbers.

More information can be found in "Concrete mathematics" by Graham, pp. 87 or visit http://mathworld.wolfram.com/CeilingFunction.html.

```
>>> from sympy import ceiling, E, I, Float, Rational
>>> ceiling(17)
17
>>> ceiling(Rational(23, 10))
3
>>> ceiling(2*E)
6
>>> ceiling(-Float(0.567))
0
>>> ceiling(I/2)
I
```

See Also:

`floor`

conjugate

Returns the complex conjugate 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

$$a - ib$$

Examples:

```
>>> from sympy.functions import conjugate
>>> from sympy import I
>>> conjugate(2)
2
>>> conjugate(I)
-I
```

class `sympy.functions.elementary.complexes.`**`conjugate`**

Changes the sign of the imaginary part of a complex number.

See Also:

`sign`, `Abs`

Examples

```
>>> from sympy import conjugate, I

>>> conjugate(1 + I)
1 - I
```

cos

class `sympy.functions.elementary.trigonometric.`**`cos`**

The cosine function.

- cos(x) -> Returns the cosine of x (measured in radians)

See Also:

`sin`, `tan`, `acos`

Notes

- cos(x) will evaluate automatically in the case x is a multiple of pi, pi/2, pi/3, pi/4 and pi/6.

References

[R51] (page 1620)

Examples

```
>>> from sympy import cos, pi
>>> from sympy.abc import x
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
>>> cos(1).diff(x)
0
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
```

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`, `tanh`, `acosh`

cot

class `sympy.functions.elementary.trigonometric.`**`cot`**

`inverse`(*argindex=1*)

Returns the inverse of this function.

coth

class sympy.functions.elementary.hyperbolic.**coth**

The hyperbolic cotangent function, $\frac{\cosh(x)}{\sinh(x)}$.

- coth(x) -> Returns the hyperbolic cotangent of x

inverse(*argindex=1*)

Returns the inverse of this function.

exp

class sympy.functions.elementary.exponential.**exp**

The exponential function, e^x.

See Also:

log

as_real_imag(*deep=True*, ***hints*)

Returns this function as a 2-tuple representing a complex number.

See Also:

sympy.functions.elementary.complexes.re (page 317), sympy.functions.elementary.complexes.im

Examples

```
>>> from sympy import I
>>> from sympy.abc import x
>>> from sympy.functions import exp
>>> exp(x).as_real_imag()
(exp(re(x))*cos(im(x)), exp(re(x))*sin(im(x)))
>>> exp(1).as_real_imag()
(E, 0)
>>> exp(I).as_real_imag()
(cos(1), sin(1))
>>> exp(1+I).as_real_imag()
(E*cos(1), E*sin(1))
```

base

Returns the base of the exponential function.

fdiff(*argindex=1*)

Returns the first derivative of this function.

static taylor_term(**args*, ***kw_args*)

Calculates the next term in the Taylor series expansion.

See Also:

classes sympy.functions.elementary.exponential.log (page 312)

ExprCondPair

class sympy.functions.elementary.piecewise.**ExprCondPair**

Represents an expression, condition pair.

cond
Returns the condition of this pair.

expr
Returns the expression of this pair.

free_symbols
Return the free symbols of this pair.

floor

class sympy.functions.elementary.integers.**floor**

Floor is a univariate function which returns the largest integer value not greater than its argument. However this implementation generalizes floor to complex numbers.

More information can be found in "Concrete mathematics" by Graham, pp. 87 or visit http://mathworld.wolfram.com/FloorFunction.html.

```
>>> from sympy import floor, E, I, Float, Rational
>>> floor(17)
17
>>> floor(Rational(23, 10))
2
>>> floor(2*E)
5
>>> floor(-Float(0.567))
-1
>>> floor(-I/2)
-I
```

See Also:

ceiling

HyperbolicFunction

class sympy.functions.elementary.hyperbolic.**HyperbolicFunction**

Base class for hyperbolic functions.

See Also:

sinh, cosh, tanh, coth

Attributes

nargs	

IdentityFunction

class sympy.functions.elementary.miscellaneous.**IdentityFunction**

The identity function

Examples

```
>>> from sympy import Id, Symbol
>>> x = Symbol('x')
>>> Id(x)
x
```

im

Returns the imaginary part of an expression.

Examples:

```
>>> from sympy.functions import im
>>> from sympy import I
>>> im(2+3*I)
3
```

See Also:

`sympy.functions.elementary.complexes.re` (page 317)

LambertW

class `sympy.functions.elementary.exponential.`**`LambertW`**

Lambert W function, defined as the inverse function of x*exp(x). This function represents the principal branch of this inverse function, which like the natural logarithm is multivalued.

For more information, see: http://en.wikipedia.org/wiki/Lambert_W_function

`fdiff`(*argindex=1*)

Return the first derivative of this function.

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`

`as_base_exp`()

Returns this function in the form (base, exponent).

`as_real_imag`(*deep=True*, ***hints*)

Returns this function as a complex coordinate.

Examples

```
>>> from sympy import I
>>> from sympy.abc import x
>>> from sympy.functions import log
>>> log(x).as_real_imag()
(log(Abs(x)), arg(x))
>>> log(I).as_real_imag()
(0, pi/2)
>>> log(1+I).as_real_imag()
(log(sqrt(2)), pi/4)
>>> log(I*x).as_real_imag()
(log(Abs(x)), arg(I*x))
```

fdiff(*argindex=1*)
Returns the first derivative of the function.

inverse(*argindex=1*)
Returns e^x, the inverse function of $\log(x)$.

static taylor_term(**args, **kw_args*)
Returns the next term in the Taylor series expansion of $\log(1+x)$.

See Also:

classes `sympy.functions.elementary.exponential.exp` (page 310)

Min

Returns the minimum of two (comparable) expressions.

Examples:

```
>>> from sympy.functions import Min
>>> Min(1,2)
1
>>> from sympy.abc import x
>>> Min(1, x)
Min(1, x)
```

It is named `Min` and not `min` to avoid conflicts with the built-in function `min`.

class `sympy.functions.elementary.miscellaneous.`**Min**
Return, if possible, the minimum value of the list.

See Also:

Max 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)

>>> Min(n, 8, p, -7, p, oo)
Min(n, -7)
```

Attributes

nargs	

Max

Returns the maximum of two (comparable) expressions

It is named `Max` and not `max` to avoid conflicts with the built-in function `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 >= 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.

See Also:

`Min` find minimum values

References

[R52] (page 1620), [R53] (page 1620)

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)

>>> 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 [R52] (page 1620).

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

Attributes

nargs	

Piecewise

class `sympy.functions.elementary.piecewise.`**`Piecewise`**

Represents a piecewise function.

Usage:

Piecewise((expr,cond), (expr,cond), ...)

- Each argument is a 2-tuple defining a 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`

Examples

```
>>> from sympy import Piecewise, log
>>> from sympy.abc import x
>>> 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)
```

Attributes

nargs	

doit(***hints*)

Evaluate this piecewise function.

sympy.functions.elementary.piecewise.**piecewise_fold**(*expr*)

Takes an expression containing a piecewise function and returns the expression in piecewise form.

See Also:

`Piecewise`

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, 1 <= x))
```

re

Return the real part of an expression.

Examples:

```
>>> from sympy.functions import re
>>> from sympy import I
>>> re(2+3*I)
2
```

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.

```
>>> 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
```

See Also:

`im`

`as_real_imag`(*deep=True*, ***hints*)

Returns the real number with a zero complex part.

See Also:

`sympy.functions.elementary.complexes.im`

root

`sympy.functions.elementary.miscellaneous.`**`root`**(*arg*, *n*)

The n-th root function (a shortcut for `arg**(1/n)`)

root(x, n) -> Returns the principal n-th root of x.

See Also:

`sympy.polys.rootoftools.RootOf` (page 1004), `sympy.core.power.integer_nthroot` (page 128), `sqrt`, `real_root`

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 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 (0,1) ]
[-1, 1]

>>> [ RootOf(x**3-1,i) for i in (0,1,2) ]
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]

>>> [ RootOf(x**4-1,i) for i in (0,1,2,3) ]
[-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 such a result real or simply return the real root in the first place:

```
>>> from sympy import real_root
>>> real_root(_)
-2
>>> real_root(-32, 5)
-2
```

RoundFunction

class sympy.functions.elementary.integers.**RoundFunction**

The base class for rounding functions.

sin

class sympy.functions.elementary.trigonometric.**sin**

The sine function.

- sin(x) -> Returns the sine of x (measured in radians)

See Also:

cos, tan, asin

Notes

- sin(x) will evaluate automatically in the case x is a multiple of pi, pi/2, pi/3, pi/4 and pi/6.

References

[R54] (page 1620)

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
```

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, tanh, asinh

as_real_imag(*deep=True*, ***hints*)

Returns this function as a complex coordinate.

fdiff(*argindex=1*)
Returns the first derivative of this function.

inverse(*argindex=1*)
Returns the inverse of this function.

static taylor_term(**args, **kw_args*)
Returns the next term in the Taylor series expansion.

sqrt

Returns the square root of an expression. It is equivalent to raise to `Rational(1,2)`.

```
>>> from sympy.functions import sqrt
>>> from sympy import Rational
>>> sqrt(2) == 2**Rational(1,2)
True
```

class sympy.functions.elementary.miscellaneous.**sqrt**
The square root function

sqrt(x) -> Returns the principal square root of x.

See Also:

sympy.polys.rootoftools.RootOf (page 1004), root

References

- http://en.wikipedia.org/wiki/Square_root
- http://en.wikipedia.org/wiki/Principal_value

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:

```
>>> sqrt(x**2).subs(x, -1)
1
>>> x.subs(x, -1)
-1
```

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)]
```

sign

class sympy.functions.elementary.complexes.**sign**

Returns the sign of an expression, that is:

- 1 if expression is positive
- 0 if expression is equal to zero
- -1 if expression is negative

See Also:

Abs, conjugate

Examples

```
>>> from sympy.functions import sign
>>> sign(-1)
-1
>>> sign(0)
0
```

tan

class sympy.functions.elementary.trigonometric.**tan**

See Also:

sin, cos, atan

Notes

- tan(x) will evaluate automatically in the case x is a multiple of pi.

References

[R55] (page 1620)

Examples

```
>>> from sympy import tan
>>> from sympy.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(1).diff(x)
0
```

inverse(*argindex=1*)
Returns the inverse of this function.

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, cosh, atanh

inverse(*argindex=1*)
Returns the inverse of this function.

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

$$B_{n,k}(x_1, x_2, \dots x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_2+\cdots=k \\ j_1+2j_2+3j_2+\cdots=n}} \frac{n!}{j_1!j_2!\cdots j_{n-k+1}!}\left(\frac{x_1}{1!}\right)^{j_1}\left(\frac{x_2}{2!}\right)^{j_2}\cdots\left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- bell(n) gives the nth Bell number, B_n.
- bell(n, x) gives the nth 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`, `catalan`, `euler`, `fibonacci`, `harmonic`, `lucas`

Notes

Not to be confused with Bernoulli numbers and Bernoulli polynomials, which use the same notation.

References

[R27] (page 1620), [R28] (page 1620), [R29] (page 1620)

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
```

Attributes

nargs	

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):

```
      n
    ___
   \      / n + 1 \
0 =  )     |       | * B .
   /___    \   k   /    k
   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:

```
          n
        ___
       \      / n \          n-k
B (x) =  )     |   | * B  * x   .
 n     /___    \ k /    k
       k = 0
```

Bernoulli numbers and Bernoulli polynomials are related as B_n(0) = B_n.

We compute Bernoulli numbers using Ramanujan's formula:

```
                          / n + 3 \
B   =  (A(n) - S(n))  /   |       |
 n                        \   n   /
```

where A(n) = (n+3)/3 when n = 0 or 2 (mod 6), A(n) = -(n+3)/6 when n = 4 (mod 6), and:

```
       [n/6]
        ___
       \      /  n + 3  \
S(n) =  )     |          | * B
       /___    \ n - 6*k /    n-6*k
       k = 1
```

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 nth Bernoulli number, B_n
- bernoulli(n, x) gives the nth Bernoulli polynomial in x, B_n(x)

See Also:

`bell`, `catalan`, `euler`, `fibonacci`, `harmonic`, `lucas`

References

[R30] (page 1620), [R31] (page 1620), [R32] (page 1620), [R33] (page 1620)

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
```

Attributes

nargs	

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)!) or C(n, k) = ff(n, k)/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 '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.

Examples

```
>>> from sympy import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True)

>>> binomial(15, 8)
6435

>>> binomial(n, -1)
0
```

```
>>> [ binomial(0, i) for i in range(1)]
[1]
>>> [ binomial(1, i) for i in range(2)]
[1, 1]
>>> [ binomial(2, i) for i in range(3)]
[1, 2, 1]
>>> [ binomial(3, i) for i in range(4)]
[1, 3, 3, 1]
>>> [ binomial(4, i) for i in range(5)]
[1, 4, 6, 4, 1]

>>> binomial(Rational(5,4), 3)
-5/128

>>> 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:

```
       1   / 2*n \
C  = ----- |     |
 n   n + 1 \  n  /
```

- catalan(n) gives the n-th Catalan number, C_n

See Also:

bell, bernoulli, euler, fibonacci, harmonic, lucas, sympy.functions.combinatorial.factorials.binomial (page 325)

References

[R34] (page 1620), [R35] (page 1620), [R36] (page 1620), [R37] (page 1620)

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 funtion 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
```

Attributes

nargs	

euler

class `sympy.functions.combinatorial.numbers.`**`euler`**

Euler numbers

The euler numbers are given by:

```
        2*n+1   k
         ___   ___                j          2*n+1
        \     \     / k \ (-1)  * (k-2*j)
E   = I  )     )    |   | --------------------
 2n     /___  /___  \ j /      k    k
        k = 1 j = 0           2  * I  * k

E     = 0
 2n+1
```

- euler(n) gives the n-th Euler number, E_n

See Also:

`bell`, `bernoulli`, `catalan`, `fibonacci`, `harmonic`, `lucas`

References

[R38] (page 1620), [R39] (page 1620), [R40] (page 1620), [R41] (page 1620)

Examples

```
>>> from sympy import Symbol, 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)
```

factorial

class `sympy.functions.combinatorial.factorials.`**`factorial`**

Implementation of factorial function over nonnegative integers. For the sake of convenience and simplicity of procedures using this function it is defined for negative integers and returns zero in this case.

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! = 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 naive product is evaluated. 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`, `RisingFactorial`, `FallingFactorial`

Examples

```
>>> from sympy import Symbol, factorial
>>> n = Symbol('n', integer=True)

>>> factorial(-2)
0

>>> factorial(0)
1

>>> factorial(7)
5040

>>> factorial(n)
factorial(n)

>>> factorial(2*n)
factorial(2*n)
```

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 integers >= -1 as:

```
       ,
      |  n*(n - 2)*(n - 4)* ... * 1    for n odd
n!! = {  n*(n - 2)*(n - 4)* ... * 2    for n even
      |  1                             for n = 0, -1
       `
```

See Also:

`factorial`, `RisingFactorial`, `FallingFactorial`

Examples

```
>>> from sympy import factorial2, var
>>> var('n')
n
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
```

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

ff(x, k) = x * (x-1) * ... * (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.

```
>>> from sympy import ff
>>> from sympy.abc import x

>>> ff(x, 0)
1

>>> ff(5, 5)
120

>>> ff(x, 5) == x*(x-1)*(x-2)*(x-3)*(x-4)
True
```

See Also:

`factorial`, `factorial2`, `RisingFactorial`

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}.

The Fibonacci polynomials are defined by F_1(x) = 1, F_2(x) = x, and F_n(x) = x*F_{n-1}(x) + F_{n-2}(x) for n > 2. For all positive integers n, F_n(1) = F_n.

- fibonacci(n) gives the nth Fibonacci number, F_n
- fibonacci(n, x) gives the nth Fibonacci polynomial in x, F_n(x)

See Also:

`bell`, `bernoulli`, `catalan`, `euler`, `harmonic`, `lucas`

References

[R42] (page 1620), [R43] (page 1620)

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
```

Attributes

nargs	

harmonic

class `sympy.functions.combinatorial.numbers.`**`harmonic`**

Harmonic numbers

The nth harmonic number is given by $\mathrm{H}_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$.

More generally:

$$\mathrm{H}_{n,m} = \sum_{k=1}^{n} \frac{1}{k^m}$$

As $n \to \infty$, $\mathrm{H}_{n,m} \to \zeta(m)$, the Riemann zeta function.

- `harmonic(n)` gives the nth harmonic number, H_n
- `harmonic(n, m)` gives the nth generalized harmonic number of order m, $\mathrm{H}_{n,m}$, where `harmonic(n) == harmonic(n, 1)`

See Also:

`bell`, `bernoulli`, `catalan`, `euler`, `fibonacci`, `lucas`

References

[R44] (page 1620), [R45] (page 1620), [R46] (page 1620)

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 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

>>> limit(harmonic(m, n), m, oo)
zeta(n)
```

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 nth Lucas number

See Also:

`bell`, `bernoulli`, `catalan`, `euler`, `fibonacci`, `harmonic`

References

[R47] (page 1620), [R48] (page 1620)

Examples

```
>>> from sympy import lucas

>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

Attributes

nargs	

MultiFactorial

class sympy.functions.combinatorial.factorials.**MultiFactorial**

Attributes

nargs	

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:

rf(x, k) = x * (x+1) * ... * (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.

See Also:

factorial, factorial2, FallingFactorial

Examples

```
>>> from sympy import rf
>>> from sympy.abc import x

>>> rf(x, 0)
1

>>> rf(1, 5)
120

>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
True
```

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:

```
{0}       {n}   {0}      {n + 1}     {n}   {  n  }
{ } = 1;  { } = { } = 0; {     } = j*{ } + {     }
{0}       {0}   {k}      {  k  }     {k}   {k - 1}
```

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 >= k >= 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 1355)

References

[R49] (page 1620), [R50] (page 1620)

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 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 != 0
4. DiracDelta(g(x)) = Sum_i(DiracDelta(x-x_i)/abs(g'(x_i))) Where x_i-s are the roots of g

Derivatives of k-th order of DiracDelta have the following property:

5. DiracDelta(x,k) = 0, for all x != 0

See Also:

Heaviside, simplify, is_simple, sympy.functions.special.tensor_functions.KroneckerDelta (page 394)

References

[R56] (page 1621)

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`, `Directdelta`

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
```

simplify(*self*, *x*)

Compute a simplified representation of the function using property number 4.

x can be:

- a symbol

See Also:

`is_simple`, `Directdelta`

Examples

```
>>> from sympy import DiracDelta
>>> from sympy.abc import x, y

>>> DiracDelta(x*y).simplify(x)
DiracDelta(x)/Abs(y)
>>> DiracDelta(x*y).simplify(y)
DiracDelta(y)/Abs(x)

>>> DiracDelta(x**2 + x - 2).simplify(x)
DiracDelta(x - 1)/3 + DiracDelta(x + 2)/3
```

Heaviside

class sympy.functions.special.delta_functions.**Heaviside**

Heaviside Piecewise function

Heaviside function has the following properties [1]:

1. **diff(Heaviside(x),x) = DiracDelta(x)** (0, if x < 0
2. **Heaviside(x) = < (1/2 if x==0 [*]** (1, if x > 0

I think is better to have H(0) = 1/2, due to the following:

```
integrate(DiracDelta(x), x) = Heaviside(x)
integrate(DiracDelta(x), (x, -oo, oo)) = 1
```

and since DiracDelta is a symmetric function, `integrate(DiracDelta(x), (x, 0, oo))` should be 1/2 (which is what Maple returns).

If we take `Heaviside(0) = 1/2`, we would have `integrate(DiracDelta(x), (x, 0, oo)) = ‘‘ ‘‘Heaviside(oo) - Heaviside(0) = 1 - 1/2 = 1/2` and `integrate(DiracDelta(x), (x, -oo, 0)) = ‘‘ ‘‘Heaviside(0) - Heaviside(-oo) = 1/2 - 0 = 1/2`

If we consider, instead `Heaviside(0) = 1`, we would have `integrate(DiracDelta(x), (x, 0, oo)) = Heaviside(oo) - Heaviside(0) = 0` and `integrate(DiracDelta(x), (x, -oo, 0)) = Heaviside(0) - Heaviside(-oo) = 1`

See Also:

`DiracDelta`

References

[R57] (page 1621)

beta

`sympy.functions.special.gamma_functions.`**`beta`**(x, y)

Euler Beta function

`beta(x, y) == gamma(x)*gamma(y) / gamma(x+y)`

Gamma and Related Functions

class `sympy.functions.special.gamma_functions.`**`gamma`**

The gamma function returns a function which passes through the integral values of the factorial function, i.e. though defined in the complex plane, when n is an integer, $\Gamma(n) = (n-1)!$

References

[R58] (page 1621)

class `sympy.functions.special.gamma_functions.`**`loggamma`**

The loggamma function is $\log(\Gamma(x))$.

[1] Regarding to the value at 0, Mathematica defines `H(0) = 1`, but Maple uses `H(0) = undefined`

References

[R59] (page 1621)

class sympy.functions.special.gamma_functions.**polygamma**

The function `polygamma(n, z)` returns `log(gamma(z)).diff(n + 1)`

$$\psi^{(n)}(z) = \frac{d^n}{dx^n} \log \Gamma(z)$$

See Also:

`gamma`, `digamma`, `trigamma`

References

[R60] (page 1621), [R61] (page 1621), [R62] (page 1621)

Examples

We can rewrite polygamma functions in terms of harmonic numbers:

```
>>> from sympy import polygamma, harmonic, Symbol
>>> x = Symbol("x")

>>> 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 logarithmic derivative of the gamma function.

In this case, `digamma(x) = polygamma(0, x)`.

See Also:

`gamma`, `trigamma`, `polygamma`

sympy.functions.special.gamma_functions.**trigamma**(x)

The trigamma function is the second of the polygamma functions.

In this case, `trigamma(x) = polygamma(1, x)`.

See Also:

`gamma`, `digamma`, `polygamma`

class sympy.functions.special.gamma_functions.**uppergamma**

Upper incomplete gamma function

It can be defined as the meromorphic continuation of

$$\Gamma(s, x) = \int_x^\infty t^{s-1} e^{-t} \mathrm{d}t = \Gamma(s) - \gamma(s, x).$$

where $\gamma(s, x)$ is the lower incomplete gamma function, `lowergamma`. 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:

$$\mathrm{E}_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\, dt = x^{n-1}\Gamma(1-n, x).$$

See Also:

`gamma`, `lowergamma`, `sympy.functions.special.hyper.hyper` (page 374)

References

[R63] (page 1621), [R64] (page 1621), [R65] (page 1621)

Examples

```
>>> from sympy import uppergamma, S
>>> from sympy.abc import s, x
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
x**2*exp(-x) + 2*x*exp(-x) + 2*exp(-x)
>>> uppergamma(-S(1)/2, x)
-2*sqrt(pi)*(-erf(sqrt(x)) + 1) + 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

$$\gamma(s, x) = \int_0^x t^{s-1} e^{-t} \mathrm{d}t.$$

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, uppergamma, sympy.functions.special.hyper.hyper (page 374)

References

[R66] (page 1621), [R67] (page 1621)

Examples

```
>>> from sympy import lowergamma, S
>>> from sympy.abc import s, x
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
-x**2*exp(-x) - 2*x*exp(-x) + 2 - 2*exp(-x)
>>> lowergamma(-S(1)/2, x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*exp(-x)/sqrt(x)
```

Special Cases of the Incomplete Gamma Functions

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:

$$\operatorname{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t} \mathrm{d}t$$

If the integral is interpreted as a Cauchy principal value, this statement holds for $x > 0$ and $\operatorname{Ei}(x)$ as defined above.

Note that we carefully avoided defining $\operatorname{Ei}(x)$ for negative real x. This is because above integral formula does not hold for any polar lift of such x, indeed all branches of $\operatorname{Ei}(x)$ above the negative reals are imaginary.

However, the following statement holds for all $x \in \mathbb{R}^*$:

$$\int_{-\infty}^{x} \frac{e^t}{t} \mathrm{d}t = \frac{\operatorname{Ei}\left(|x| e^{i \arg(x)}\right) + \operatorname{Ei}\left(|x| e^{-i \arg(x)}\right)}{2},$$

where the integral is again understood to be a principal value if $x > 0$, and $|x|e^{i\arg(x)}$, $|x|e^{-i\arg(x)}$ denote two conjugate polar lifts of x.

See Also:

expint (page 343) Generalised exponential integral.

E1 (page 344) Special case of the generalised exponential integral.

li (page 345) Logarithmic integral.

Li (page 346) Offset logarithmic integral.

Si (page 347) Sine integral.

Ci (page 348) Cosine integral.

Shi (page 349) Hyperbolic sine integral.

Chi (page 350) Hyperbolic cosine integral.

sympy.functions.special.gamma_functions.uppergamma (page 339)

References

[R68] (page 1621), [R69] (page 1621), [R70] (page 1621)

Examples

```
>>> from sympy import Ei, polar_lift, exp_polar, I, pi
>>> from sympy.abc import x
```

The exponential integral in SymPy is strictly undefined for negative values of the argument. For convenience, exponential integrals with negative arguments are immediately converted into an expression that agrees with the classical integral definition:

```
>>> Ei(-1)
-I*pi + Ei(exp_polar(I*pi))
```

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

$$\mathrm{E}_\nu(z) = z^{\nu-1}\Gamma(1-\nu, z),$$

where $\Gamma(1-\nu, z)$ is the upper incomplete gamma function (uppergamma).

Hence for z with positive real part we have

$$\mathrm{E}_\nu(z) = \int_1^\infty \frac{e^{-zt}}{z^\nu}\mathrm{d}t,$$

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for $\mathrm{E}_\nu(z)$. 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 341) Another related function called exponential integral.

E1 (page 344) The classical case, returns expint(1, z).

li (page 345) Logarithmic integral.

Li (page 346) Offset logarithmic integral.

Si (page 347) Sine integral.

Ci (page 348) Cosine integral.

Shi (page 349) Hyperbolic sine integral.

Chi (page 350) Hyperbolic cosine integral.

sympy.functions.special.gamma_functions.uppergamma (page 339)

References

[R71] (page 1621), [R72] (page 1621), [R73] (page 1621)

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 nu has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg((), (1, 1)), ((0, 0, -nu + 1), ()), z)
```

At non-postive 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)*erf(sqrt(z))/sqrt(z) + sqrt(pi)/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.`**`E1`**`(z)`

Classical case of the generalized exponential integral.

This is equivalent to `expint(1, z)`.

See Also:

Ei (page 341) Exponential integral.

expint (page 343) Generalised exponential integral.

li (page 345) Logarithmic integral.

Li (page 346) Offset logarithmic integral.

Si (page 347) Sine integral.

Ci (page 348) Cosine integral.

Shi (page 349) Hyperbolic sine integral.

Chi (page 350) Hyperbolic cosine integral.

class `sympy.functions.special.error_functions.li`

The classical logarithmic integral.

For the use in SymPy, this function is defined as

$$\operatorname{li}(x) = \int_0^x \frac{1}{\log(t)} \mathrm{d}t \,.$$

See Also:

Li (page 346) Offset logarithmic integral.

Ei (page 341) Exponential integral.

expint (page 343) Generalised exponential integral.

E1 (page 344) Special case of the generalised exponential integral.

Si (page 347) Sine integral.

Ci (page 348) Cosine integral.

Shi (page 349) Hyperbolic sine integral.

Chi (page 350) Hyperbolic cosine integral.

References

[R74] (page 1621), [R75] (page 1621), [R76] (page 1621), [R77] (page 1621)

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)
1.06527957843574982470011255 98 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```
>>> from sympy.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

$$\mathrm{Li}(x) = \mathrm{li}(x) - \mathrm{li}(2)$$

See Also:

`li` (page 345) Logarithmic integral.

`Ei` (page 341) Exponential integral.

`expint` (page 343) Generalised exponential integral.

`El` (page 344) Special case of the generalised exponential integral.

`Si` (page 347) Sine integral.

`Ci` (page 348) Cosine integral.

`Shi` (page 349) Hyperbolic sine integral.

`Chi` (page 350) Hyperbolic cosine integral.

References

[R78] (page 1621), [R79] (page 1621), [R80] (page 1621)

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

$$\operatorname{Si}(z) = \int_0^z \frac{\sin t}{t} \mathrm{d}t.$$

It is an entire function.

See Also:

`Ci` (page 348) Cosine integral.

`Shi` (page 349) Hyperbolic sine integral.

`Chi` (page 350) Hyperbolic cosine integral.

`Ei` (page 341) Exponential integral.

`expint` (page 343) Generalised exponential integral.

`E1` (page 344) Special case of the generalised exponential integral.

`li` (page 345) Logarithmic integral.

`Li` (page 346) Offset logarithmic integral.

References

[R81] (page 1621)

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
```

class `sympy.functions.special.error_functions.`**`Ci`**

Cosine integral.

This function is defined for positive x by

$$\operatorname{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} \mathrm{d}t = -\int_x^\infty \frac{\cos t}{t} \mathrm{d}t,$$

where γ is the Euler-Mascheroni constant.

We have

$$\operatorname{Ci}(z) = -\frac{\operatorname{E}_1\left(e^{i\pi/2} z\right) + \operatorname{E}_1\left(e^{-i\pi/2} z\right)}{2}$$

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 347) Sine integral.

Shi (page 349) Hyperbolic sine integral.

Chi (page 350) Hyperbolic cosine integral.

Ei (page 341) Exponential integral.

expint (page 343) Generalised exponential integral.

E1 (page 344) Special case of the generalised exponential integral.

li (page 345) Logarithmic integral.

Li (page 346) Offset logarithmic integral.

References

[R82] (page 1621)

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

$$\operatorname{Shi}(z) = \int_0^z \frac{\sinh t}{t} \mathrm{d}t.$$

It is an entire function.

See Also:

Si (page 347) Sine integral.

Ci (page 348) Cosine integral.

Chi (page 350) Hyperbolic cosine integral.

Ei (page 341) Exponential integral.

expint (page 343) Generalised exponential integral.

E1 (page 344) Special case of the generalised exponential integral.

li (page 345) Logarithmic integral.

Li (page 346) Offset logarithmic integral.

References

[R83] (page 1621)

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

$$\operatorname{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} \mathrm{d}t,$$

where γ is the Euler-Mascheroni constant.

We have

$$\operatorname{Chi}(z) = \operatorname{Ci}\left(e^{i\pi/2}z\right) - i\frac{\pi}{2},$$

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 347) Sine integral.

Ci (page 348) Cosine integral.

Shi (page 349) Hyperbolic sine integral.

Ei (page 341) Exponential integral.

expint (page 343) Generalised exponential integral.

E1 (page 344) Special case of the generalised exponential integral.

li (page 345) Logarithmic integral.

Li (page 346) Offset logarithmic integral.

References

[R84] (page 1621)

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
```

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

$$\mathrm{S}(z) = \int_0^z \sin\frac{\pi}{2}t^2 \mathrm{d}t.$$

It is an entire function.

See Also:

fresnelc (page 353)

References

[R85] (page 1621), [R86] (page 1621), [R87] (page 1621), [R88] (page 1621)

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

$$\mathrm{C}(z) = \int_0^z \cos\frac{\pi}{2}t^2 \mathrm{d}t.$$

It is an entire function.

See Also:

fresnels (page 352)

References

[R89] (page 1621), [R90] (page 1621), [R91] (page 1621), [R92] (page 1621)

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
```

Error Functions

class `sympy.functions.special.error_functions.`**`erf`**

The Gauss error function.

This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}x.$$

Or, in ASCII:

```
        x
    /
   |
   |     2
   |   -t
2* |  e    dt
```

```
   |
  /
  0
-------------
    ____
    \/ pi
```

See Also:

erfc (page 356) Complementary error function.

erfi (page 357) Imaginary error function.

erf2 (page 358) Two-argument error function.

erfinv (page 359) Inverse error function.

erfcinv (page 360) Inverse Complementary error function.

erf2inv (page 360) Inverse two-argument error function.

References

[R93] (page 1622), [R94] (page 1622), [R95] (page 1622), [R96] (page 1622)

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)
-1296959.730717639231527940950 62*I
```

class `sympy.functions.special.error_functions.`**`erfc`**

Complementary Error Function. The function is defined as:

$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} \mathrm{d}t$

See Also:

`erf` **(page 354)** Gaussian error function.

`erfi` **(page 357)** Imaginary error function.

`erf2` **(page 358)** Two-argument error function.

`erfinv` **(page 359)** Inverse error function.

`erfcinv` **(page 360)** Inverse Complementary error function.

`erf2inv` **(page 360)** Inverse two-argument error function.

References

[R97] (page 1622), [R98] (page 1622), [R99] (page 1622), [R100] (page 1622)

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)
0.000000015417257900280018852159673486 9

>>> erfc(4*I).evalf(30)
1.0 - 1296959.730717639231527940950 62*I
```

class `sympy.functions.special.error_functions.`**`erfi`**

Imaginary error function. The function erfi is defined as:

$\mathrm{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} \mathrm{d}t$

See Also:

`erf` (page 354) Gaussian error function.

`erfc` (page 356) Complementary error function.

`erf2` (page 358) Two-argument error function.

`erfinv` (page 359) Inverse error function.

`erfcinv` (page 360) Inverse Complementary error function.

`erf2inv` (page 360) Inverse two-argument error function.

References

[R101] (page 1622), [R102] (page 1622), [R103] (page 1622)

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)
18.564802414575552598704291913 2
```

```
>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

class `sympy.functions.special.error_functions.`**`erf2`**

Two-argument error function. This function is defined as:

$\mathrm{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} \mathrm{d}t$

See Also:

`erf` **(page 354)** Gaussian error function.

`erfc` **(page 356)** Complementary error function.

`erfi` **(page 357)** Imaginary error function.

`erfinv` **(page 359)** Inverse error function.

`erfcinv` **(page 360)** Inverse Complementary error function.

`erf2inv` **(page 360)** Inverse two-argument error function.

References

[R104] (page 1622)

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
>>> 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:

$\mathrm{erf}(x) = y \quad \Rightarrow \quad \mathrm{erfinv}(y) = x$

See Also:

`erf` **(page 354)** Gaussian error function.

`erfc` **(page 356)** Complementary error function.

`erfi` **(page 357)** Imaginary error function.

`erf2` **(page 358)** Two-argument error function.

`erfcinv` **(page 360)** Inverse Complementary error function.

`erf2inv` **(page 360)** Inverse two-argument error function.

References

[R105] (page 1622), [R106] (page 1622)

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:

$\mathrm{erfc}(x) = y \quad \Rightarrow \quad \mathrm{erfcinv}(y) = x$

See Also:

`erf` **(page 354)** Gaussian error function.

`erfc` **(page 356)** Complementary error function.

`erfi` **(page 357)** Imaginary error function.

`erf2` **(page 358)** Two-argument error function.

`erfinv` **(page 359)** Inverse error function.

`erf2inv` **(page 360)** Inverse two-argument error function.

References

[R107] (page 1622), [R108] (page 1622)

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:

$\text{erf2}(x, w) = y \quad \Rightarrow \quad \text{erf2inv}(x, y) = w$

See Also:

erf (page 354) Gaussian error function.

erfc (page 356) Complementary error function.

erfi (page 357) Imaginary error function.

erf2 (page 358) Two-argument error function.

erfinv (page 359) Inverse error function.

erfcinv (page 360) Inverse complementary error function.

References

[R109] (page 1622)

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
```

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

$$z^2 \frac{\mathrm{d}^2 w}{\mathrm{d}z^2} + z \frac{\mathrm{d}w}{\mathrm{d}z} + (z^2 - \nu^2) w = 0,$$

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`, `besseli`, `besselk`

References

[R110] (page 1622), [R111] (page 1622), [R112] (page 1622), [R113] (page 1622)

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

$$Y_\nu(z) = \lim_{\mu\to\nu} \frac{J_\mu(z)\cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

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, besseli, besselk

References

[R114] (page 1622)

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

$$z^2\frac{\mathrm{d}^2w}{\mathrm{d}z^2} + z\frac{\mathrm{d}w}{\mathrm{d}z} + (z^2+\nu^2)^2 w = 0.$$

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, bessely, besselk

References

[R115] (page 1622)

Examples

```
>>> from sympy import besseli
>>> from sympy.abc import z, n
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(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

$$K_\nu(z) = \lim_{\mu\to\nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_\mu(z)}{\sin(\pi\mu)},$$

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, besseli, bessely

References

[R116] (page 1622)

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, besselj, bessely

References

[R117] (page 1622)

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`, `besselj`, `bessely`

References

[R118] (page 1622)

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

$$z^2 \frac{\mathrm{d}^2 w}{\mathrm{d}z^2} + 2z \frac{\mathrm{d}w}{\mathrm{d}z} + (z^2 - \nu(\nu + 1))w = 0.$$

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.

See Also:

besselj, bessely, besselk, yn

Examples

```
>>> from sympy import Symbol, jn, sin, cos, expand_func
>>> z = Symbol("z")
>>> print jn(0, z).expand(func=True)
sin(z)/z
>>> jn(1, z).expand(func=True) == 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)
```

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()`.

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

$$j_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+\frac{1}{2}}(z),$$

where $Y_\nu(z)$ is the Bessel function of the second kind.

See Also:

besselj, bessely, besselk, jn

Examples

```
>>> from sympy import Symbol, yn, sin, cos, expand_func
>>> z = Symbol("z")
>>> print expand_func(yn(0, z))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2-sin(z)/z
True
```

For integral orders n, y_n is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

`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()` (page 727)
- 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`, `yn`, `besselj`, `besselk`, `bessely`

Examples

```
>>> from sympy import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

B-Splines

`sympy.functions.special.bsplines.`**`bspline_basis`**(*d, knots, n, x, close=True*)

The n-th B-spline at x of degree d with knots.

B-Splines are piecewise polynomials of degree d [R119] (page 1622). 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, And(x <= 1, x >= 0)), (0, True))
```

For a given `(d, knots)` there are `len(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, And(x < 1, x >= 0)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3, And(x < 2, x >= 1)),
          (x**3/2 - 4*x**2 + 10*x - 22/3, And(x < 3, x >= 2)),
          (-x**3/6 + 2*x**2 - 8*x + 32/3, And(x <= 4, x >= 3)),
          (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, And(x <= 2, x >= 0)), (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

[R119] (page 1622)

sympy.functions.special.bsplines.**bspline_basis_set**(*d, knots, x*)

Return the len(knots)-d-1 B-splines at x of degree d with knots.

This function returns a list of Piecewise polynomials that are the len(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, And(x < 1, x >= 0)),
           (-x**2 + 3*x - 3/2, And(x < 2, x >= 1)),
           (x**2/2 - 3*x + 9/2, And(x <= 3, x >= 2)),
           (0, True)),
 Piecewise((x**2/2 - x + 1/2, And(x < 2, x >= 1)),
           (-x**2 + 5*x - 11/2, And(x < 3, x >= 2)),
           (x**2/2 - 4*x + 8, And(x <= 4, x >= 3)),
           (0, True))]
```

Riemann Zeta and Related Functions

class sympy.functions.special.zeta_functions.**zeta**

Hurwitz zeta function (or Riemann zeta function).

For $\mathrm{Re}(a) > 0$ and $\mathrm{Re}(s) > 1$, 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 $\mathrm{Re}(a) > 0$ 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 $\mathrm{Re}(a) < 0$), see the documentation of `lerchphi` (page 372) 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 370), `lerchphi` (page 372), `polylog` (page 371)

References

[R120] (page 1622), [R121] (page 1622)

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 $\mathrm{Re}(s) > 0$, 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 368)

References

[R122] (page 1622)

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

$$\mathrm{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

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

$$\mathrm{Li}_{s+1}(z) = \int_0^z \frac{\mathrm{Li}_s(t)}{t} \mathrm{d}t.$$

The polylogarithm is a special case of the Lerch transcendent:

$$\mathrm{Li}_s(z) = z\Phi(z, s, 1)$$

See Also:

zeta (page 368), lerchphi (page 372)

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*exp_polar(-I*pi) + 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 $\operatorname{Re}(a) > 0$, $|z| < 1$ 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 $\operatorname{Re}(a) \le 0$.

Assume now $\operatorname{Re}(a) > 0$. The integral representation

$$\Phi_0(z, s, a) = \int_0^\infty \frac{t^{s-1} e^{-at}}{1 - z e^{-t}} \frac{\mathrm{d}t}{\Gamma(s)}$$

provides an analytic continuation to $\mathbb{C} - [1, \infty)$. Finally, for $x \in (1, \infty)$ we find

$$\lim_{\epsilon \to 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \to 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},$$

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 371), zeta (page 368)

References

[R123] (page 1622), [R124] (page 1622), [R125] (page 1622)

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)
```

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

$$ {}_pF_q\left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix}\middle| z\right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n}{(b_1)_n \dots (b_q)_n} \frac{z^n}{n!}, $$

where $(a)_n = (a)(a+1)\dots(a+n-1)$ 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 $p \le q$, 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 1201), `sympy.functions.special.gamma_functions.gamma` (page 338), `meijerg`

References

[R126] (page 1622), [R127] (page 1622)

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 oo, 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*" $a_1, \ldots, a_n$ and $a_{n+1}, \ldots, a_p$, and there are "*denominator parameters*" $b_1, \ldots, b_m$ and $b_{m+1}, \ldots, b_q$. 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):

$$G_{p,q}^{m,n} \left(\begin{matrix} a_1, \ldots, a_n & a_{n+1}, \ldots, a_p \\ b_1, \ldots, b_m & b_{m+1}, \ldots, b_q \end{matrix} \middle| z \right).$$

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:

$$\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s \mathrm{d}s,$$

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 $j \le n$ and $k \le m$.

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`, `sympy.simplify.hyperexpand` (page 1201)

References

[R128] (page 1622), [R129] (page 1622)

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.bother
(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))
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*exp(I*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).

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

$$K(z) = F\left(\frac{\pi}{2}\middle|z\right)$$

where $F\left(z|m\right)$ is the Legendre incomplete elliptic integral of the first kind.

The function $K(z)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

See Also:

elliptic_f (page 379)

References

[R130] (page 1623), [R131] (page 1623)

Examples

```
>>> from sympy import elliptic_k, I, pi
>>> from sympy.abc import z
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
```

class sympy.functions.special.elliptic_integrals.**elliptic_f**

The Legendre incomplete elliptic integral of the first kind, defined by

$$F\left(z|m\right) = \int_0^z \frac{dt}{\sqrt{1 - m\sin^2 t}}$$

This function reduces to a complete elliptic integral of the first kind, $K(m)$, when $z = \pi/2$.

See Also:

elliptic_k (page 378)

References

[R132] (page 1623), [R133] (page 1623)

Examples

```
>>> from sympy import elliptic_f, I, O
>>> from sympy.abc import z, m
>>> elliptic_f(z, m).series(z)
z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + O(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

$$E\left(z|m\right) = \int_0^z \sqrt{1 - m\sin^2 t}dt$$

Called with a single argument z, evaluates the Legendre complete elliptic integral of the second kind

$$E(z) = E\left(\frac{\pi}{2}\middle|z\right)$$

The function $E(z)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

References

[R134] (page 1623), [R135] (page 1623), [R136] (page 1623)

Examples

```
>>> from sympy import elliptic_e, I, pi, O
>>> from sympy.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + O(z**6)
>>> 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

$$\Pi\left(n; z\middle|m\right) = \int_0^z \frac{dt}{\left(1 - n\sin^2 t\right)\sqrt{1 - m\sin^2 t}}$$

Called with two arguments n and m, evaluates the complete elliptic integral of the third kind:

$$\Pi\left(n\middle|m\right) = \Pi\left(n; \frac{\pi}{2}\middle|m\right)$$

References

[R137] (page 1623), [R138] (page 1623), [R139] (page 1623)

Examples

```
>>> from sympy import elliptic_pi, I, pi, O, S
>>> from sympy.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z)
z + z**3*(m/6 + n/3) + z**5*(3*m**2/40 + m*n/10 - m/30 + n**2/5 - n/15) + O(z**6)
```

```
>>> 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
```

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 nth 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 383), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R140] (page 1623), [R141] (page 1623), [R142] (page 1623)

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 nth 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:

$$\int_{-1}^{1} P_m^{(\alpha,\beta)}(x) P_n^{(\alpha,\beta)}(x)(1-x)^\alpha (1+x)^\beta \mathrm{d}x = \delta_{m,n}$$

See Also:

gegenbauer (page 383), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R143] (page 1623), [R144] (page 1623), [R145] (page 1623)

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 + 2*n + 1)*fact
```

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 $\left(1-x^2\right)^{\alpha-\frac{1}{2}}$.

See Also:

jacobi (page 381), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R146] (page 1623), [R147] (page 1623), [R148] (page 1623)

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 nth 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 381), gegenbauer (page 383), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R149] (page 1623), [R150] (page 1623), [R151] (page 1623), [R152] (page 1623), [R153] (page 1623)

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 nth 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 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R154] (page 1623), [R155] (page 1623), [R156] (page 1623), [R157] (page 1623), [R158] (page 1623)

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)
(-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**

chebyshev_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if 0 <= k < n, chebyshevt(n, chebyshevt_root(n, k)) == 0.

See Also:

jacobi (page 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

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 <= k < n, chebyshevu(n, chebyshevu_root(n, k)) == 0.

See Also:

chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu (page 385), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

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 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R159] (page 1623), [R160] (page 1623), [R161] (page 1623), [R162] (page 1623)

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:

$$P_n^m(x) = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{\mathrm{d}^m P_n(x)}{\mathrm{d}x^m}$$

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 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), hermite (page 388), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly

(page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R163] (page 1623), [R164] (page 1623), [R165] (page 1623), [R166] (page 1623)

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 nth Hermite polynomial in x, $H_n(x)$

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight $\exp\left(-\frac{x^2}{2}\right)$.

See Also:

jacobi (page 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), laguerre (page 389), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R167] (page 1623), [R168] (page 1624), [R169] (page 1624)

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)
>>> 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 nth Laguerre polynomial in x, $L_n(x)$.

Parameters **n** : int

Degree of Laguerre polynomial. Must be `n >= 0`.

See Also:

jacobi (page 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), assoc_laguerre (page 390), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R170] (page 1624), [R171] (page 1624), [R172] (page 1624), [R173] (page 1624)

Examples

```
>>> from sympy import laguerre, diff
>>> from sympy.abc import x, n
>>> laguerre(0, x)
1
>>> 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 nth generalized Laguerre polynomial in x, $L_n(x)$.

Parameters **n** : int

Degree of Laguerre polynomial. Must be `n >= 0`.

alpha : Expr

Arbitrary expression. For `alpha=0` regular Laguerre polynomials will be generated.

See Also:

jacobi (page 381), gegenbauer (page 383), chebyshevt (page 384), chebyshevt_root (page 386), chebyshevu (page 385), chebyshevu_root (page 386), legendre (page 386), assoc_legendre (page 387), hermite (page 388), laguerre (page 389), sympy.polys.orthopolys.jacobi_poly (page 1006), sympy.polys.orthopolys.gegenbauer_poly (page 1006), sympy.polys.orthopolys.chebyshevt_poly (page 1006), sympy.polys.orthopolys.chebyshevu_pol (page 1006), sympy.polys.orthopolys.hermite_poly (page 1006), sympy.polys.orthopolys.legendre_poly (page 1006), sympy.polys.orthopolys.laguerre_poly (page 1006)

References

[R174] (page 1624), [R175] (page 1624), [R176] (page 1624), [R177] (page 1624)

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

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}}\exp(im\varphi)\mathrm{P}_n^m\left(\cos(\theta)\right)$$

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`, `Znm`

References

[R178] (page 1624), [R179] (page 1624), [R180] (page 1624), [R181] (page 1624)

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 evalute 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,

>>> 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))
```

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, Znm

References

[R182] (page 1624), [R183] (page 1624), [R184] (page 1624)

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, Ynm_c

References

[R185] (page 1624), [R186] (page 1624), [R187] (page 1624)

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`

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`

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
```

Attributes

nargs	

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`, `sympy.functions.special.delta_functions.DiracDelta` (page 336)

References

[R188] (page 1624)

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`, `is_only_below_fermi`, `is_only_above_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, 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`, `is_only_above_fermi`, `is_only_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, 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, is_below_fermi, is_only_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, 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, is_below_fermi, is_only_above_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, 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

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`

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
```

5.9 Geometric Algebra Module

A module for geometric algebra

Documentation for Geometric Algebra module.

Contents:

5.9.1 Geometric Algebra

Author Alan Bromborsky

Abstract

This document describes the implementation, installation and use of a geometric algebra module written in python that utilizes the `sympy` (page 612) symbolic algebra library. The python module *galgebra* has been developed for coordinate free calculations using the operations (geometric, outer, and inner products etc.) of geometric algebra. The operations can be defined using a completely arbitrary metric defined by the inner products of a set of arbitrary vectors or the metric can be restricted to enforce orthogonality and signature constraints on the set of vectors. In addition the module includes the geometric, outer (curl) and inner (div) derivatives and the ability to define a curvilinear coordinate system. The module requires the sympy module and the numpy module for numerical linear algebra calculations. For latex output a latex distribution must be installed.

What is Geometric Algebra?

Geometric algebra is the Clifford algebra of a real finite dimensional vector space or the algebra that results when a real finite dimensional vector space is extended with a product of vectors (geometric product) that is associative, left and right distributive, and yields a real number for the square (geometric product) of any vector [Hestenes] (page 1624), [Doran] (page 1624). The elements of the geometric algebra are called multivectors and consist of the linear combination of scalars, vectors, and the geometric product of two or more vectors. The additional axioms for the geometric algebra are that for any vectors a, b, and c in the base vector space ([Doran] (page 1624),p85):

$$\begin{gathered} a\left(bc\right) = \left(ab\right)c \\ a\left(b+c\right) = ab + ac \\ \left(a+b\right)c = ac + bc \\ aa = a^2 \in \Re \end{gathered}$$

By induction the first three axioms also apply to any multivectors. The dot product of two vectors is defined by ([Doran] (page 1624),p86)

$$a \cdot b \equiv \left(ab + ba\right)/2$$

Then consider

$$\begin{aligned} c &= a + b \\ c^2 &= \left(a+b\right)^2 \\ c^2 &= a^2 + ab + ba + b^2 \\ a \cdot b &= \left(c^2 - a^2 - b^2\right)/2 \in \Re \end{aligned}$$

Thus $a \cdot b$ is real. The objects generated from linear combinations of the geometric products of vectors are called multivectors. If a basis for the underlying vector space is the set of vectors formed from $e_1, \ldots, e_n$ a complete basis for the geometric algebra is given by the scalar 1, the vectors $e_1, \ldots, e_n$ and all geometric products of vectors

$$e_{i_1} e_{i_2} \ldots e_{i_r} \text{ where } 0 \leq r \leq n,\ 0 \leq i_j \leq n \text{ and } 0 < i_1 < i_2 < \cdots < i_r \leq n$$

Each base of the complete basis is represented by a noncommutative symbol (except for the scalar 1) with name $e_{i_1} \ldots e_{i_r}$ so that the general multivector $\boldsymbol{A}$ is represented by (A is the scalar part of the multivector and the $A^{i_1,\ldots,i_r}$ are scalars)

$$\boldsymbol{A} = A + \sum_{r=1}^{n} \sum_{i_1,\ldots,i_r,\ \forall\ 0 \leq i_j \leq n} A^{i_1,\ldots,i_r} e_{i_1} e_{i_2} \ldots e_r$$

The critical operation in setting up the geometric algebra is reducing the geometric product of any two bases to a linear combination of bases so that we can calculate a multiplication table for the bases. Since the geometric product is associative we can use the operation (by definition for two vectors $a \cdot b \equiv (ab + ba)/2$ which is a scalar)

$$e_{i_{j+1}} e_{i_j} = 2 e_{i_{j+1}} \cdot e_{i_j} - e_{i_j} e_{i_{j+1}} \tag{5.1}$$

These processes are repeated untill every basis list in $\boldsymbol{A}$ is in normal (ascending) order with no repeated elements. As an example consider the following

$$\begin{aligned} e_3 e_2 e_1 &= (2(e_2 \cdot e_3) - e_2 e_3) e_1 \\ &= 2(e_2 \cdot e_3) e_1 - e_2 e_3 e_1 \\ &= 2(e_2 \cdot e_3) e_1 - e_2 (2(e_1 \cdot e_3) - e_1 e_3) \\ &= 2((e_2 \cdot e_3) e_1 - (e_1 \cdot e_3) e_2) + e_2 e_1 e_3 \\ &= 2((e_2 \cdot e_3) e_1 - (e_1 \cdot e_3) e_2 + (e_1 \cdot e_2) e_3) - e_1 e_2 e_3 \end{aligned}$$

which results from repeated application of equation *5.1* (page 400). If the product of basis vectors contains repeated factors equation *5.1* (page 400) can be used to bring the repeated factors next to one another so that if $e_{i_j} = e_{i_{j+1}}$ then $e_{i_j} e_{i_{j+1}} = e_{i_j} \cdot e_{i_{j+1}}$ which is a scalar that commutes with all the terms in the product and can be brought to the front of the product. Since every repeated pair of vectors in a geometric product of r factors reduces the number of noncommutative factors in the product by $r - 2$. The number of bases in the multivector algebra is 2^n and the number containing r factors is $\binom{n}{r}$ which is the number of combinations or n things taken r at a time (binominal coefficient).

The other construction required for formulating the geometric algebra is the outer or wedge product (symbol $\wedge$) of r vectors denoted by $a_1 \wedge \cdots \wedge a_r$. The wedge product of r vectors is called an r-blade and is defined by ([Doran] (page 1624),p86)

$$a_1 \wedge \cdots \wedge a_r \equiv \sum_{i_{j_1} \ldots i_{j_r}} \epsilon^{i_{j_1} \ldots i_{j_r}} a_{i_{j_1}} \ldots a_{i_{j_1}}$$

where $\epsilon^{i_{j_1} \ldots i_{j_r}}$ is the contravariant permutation symbol which is $+1$ for an even permutation of the superscripts, 0 if any superscripts are repeated, and -1 for an odd permutation of the superscripts. From the definition $a_1 \wedge \cdots \wedge a_r$ is antisymmetric in all its arguments and the following relation for the wedge product of a vector a and an r-blade B_r can be derived

$$a \wedge B_r = (a B_r + (-1)^r B_r a)/2 \tag{5.2}$$

Using equation *5.2* (page 400) one can represent the wedge product of all the basis vectors in terms of the geometric product of all the basis vectors so that one can solve (the system of equations is lower diagonal) for the geometric product of all the basis vectors in terms of the wedge product of all the basis vectors. Thus a general multivector $\boldsymbol{B}$ can be represented as a linear combination of a scalar and the basis blades.

$$\boldsymbol{B} = B + \sum_{r=1}^{n} \sum_{i_1, \ldots, i_r, \, \forall \, 0 \le i_j \le n} B^{i_1, \ldots, i_r} e_{i_1} \wedge e_{i_2} \wedge \cdots \wedge e_r$$

Using the blades $e_{i_1} \wedge e_{i_2} \wedge \cdots \wedge e_r$ creates a graded algebra where r is the grade of the basis blades. The grade-r part of $\boldsymbol{B}$ is the linear combination of all terms with grade r basis blades. The scalar part of $\boldsymbol{B}$ is defined to be grade-0. Now that the blade expansion of $\boldsymbol{B}$ is defined we can also define the grade projection operator $\langle \boldsymbol{B} \rangle_r$ by

$$\langle \boldsymbol{B} \rangle_r = \sum_{i_1, \ldots, i_r, \, \forall \, 0 \le i_j \le n} B^{i_1, \ldots, i_r} e_{i_1} \wedge e_{i_2} \wedge \cdots \wedge e_r$$

and

$$\langle \boldsymbol{B} \rangle \equiv \langle \boldsymbol{B} \rangle_0 = B$$

Then if $\boldsymbol{A}_r$ is an r-grade multivector and $\boldsymbol{B}_s$ is an s-grade multivector we have

$$\boldsymbol{A}_r \boldsymbol{B}_s = \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{|r-s|} + \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{|r-s|+2} + \cdots \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{r+s}$$

and define ([Hestenes] (page 1624),p6)

$$\begin{aligned} \boldsymbol{A}_r \wedge \boldsymbol{B}_s &\equiv \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{r+s} \\ \boldsymbol{A}_r \cdot \boldsymbol{B}_s &\equiv \left\{ \begin{array}{ll} r \text{ or } s \neq 0: & \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{|r-s|} \\ r \text{ or } s = 0: & 0 \end{array} \right\} \end{aligned}$$

where $\boldsymbol{A}_r \cdot \boldsymbol{B}_s$ is called the dot or inner product of two pure grade multivectors. For the case of two non-pure grade multivectors

$$\begin{aligned} \boldsymbol{A} \wedge \boldsymbol{B} &= \sum_{r,s} \langle \boldsymbol{A} \rangle_r \wedge \langle \boldsymbol{B} \rangle_s \\ \boldsymbol{A} \cdot \boldsymbol{B} &= \sum_{r,s \neq 0} \langle \boldsymbol{A} \rangle_r \cdot \langle \boldsymbol{B} \rangle_s \end{aligned}$$

Two other products, the right ($\rfloor$) and left ($\lfloor$) contractions, are defined by

$$\begin{aligned} \boldsymbol{A} \lfloor \boldsymbol{B} &\equiv \sum_{r,s} \left\{ \begin{array}{ll} \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{r-s} & r \geq s \\ 0 & r < s \end{array} \right\} \\ \boldsymbol{A} \rfloor \boldsymbol{B} &\equiv \sum_{r,s} \left\{ \begin{array}{ll} \langle \boldsymbol{A}_r \boldsymbol{B}_s \rangle_{s-r} & s \geq r \\ 0 & s < r \end{array} \right\} \end{aligned}$$

A final operation for multivectors is the reverse. If a multivector $\boldsymbol{A}$ is the geometric product of r vectors (versor) so that $\boldsymbol{A} = a_1 \ldots a_r$ the reverse is defined by

$$\boldsymbol{A}^\dagger \equiv a_r \ldots a_1$$

where for a general multivector we have (the the sum of the reverse of versors)

$$\boldsymbol{A}^\dagger = A + \sum_{r=1}^{n} (-1)^{r(r-1)/2} \sum_{i_1,\ldots,i_r,\ \forall\ 0 \leq i_j \leq n} A^{i_1,\ldots,i_r} e_{i_1} \wedge e_{i_2} \wedge \cdots \wedge e_r$$

note that if $\boldsymbol{A}$ is a versor then $\boldsymbol{A}\boldsymbol{A}^\dagger \in \Re$ and ($AA^\dagger \neq 0$)

$$\boldsymbol{A}^{-1} = \frac{\boldsymbol{A}^\dagger}{\boldsymbol{A}\boldsymbol{A}^\dagger}$$

Representation of Multivectors in Sympy

The sympy python module offers a simple way of representing multivectors using linear combinations of commutative expressions (expressions consisting only of commuting sympy objects) and noncommutative symbols. We start by defining n noncommutative sympy symbols

```
(e_1,...,e_n) = symbols('e_1,...,e_n',commutative=False)
```

Several software packages for numerical geometric algebra calculations are available from Doran-Lasenby group and the Dorst group. Symbolic packages for Clifford algebra using orthongonal bases such as $e_i e_j + e_j e_i = 2\eta_{ij}$, where η_{ij} is a numeric array are available in Maple and Mathematica. The symbolic algebra module, *galgebra*, developed for python does not depend on an orthogonal basis representation, but rather is generated from a set of n arbitrary symbolic vectors, $e_1, e_2, \ldots, e_n$ and a symbolic metric tensor $g_{ij} = e_i \cdot e_j$.

In order not to reinvent the wheel all scalar symbolic algebra is handled by the python module `sympy` (page 612) and the abstract basis vectors are encoded as noncommuting sympy symbols.

The basic geometic algebra operations will be implemented in python by defining a multivector class, MV, and overloading the python operators in Table *5.1* (page 402) where *A* and *B* are any two multivectors (In the case of +, -, *, ^, and | the operation is also defined if *A* or *B* is a sympy symbol or a sympy real number).

Operation	Result
''A+B''	sum of multivectors
''A-B''	difference of multivectors
''A*B''	geometric product
''A^B''	outer product of multivectors
''A|B''	inner product of multivectors
''A<B''	left contraction of multivectors
''A>B''	right contraction of multivectors

Table *5.1* (page 402). Multivector operations for *galgebra*

Since < and > have no r-forms (in python for the < and > operators there are no *__rlt__()* and *__rlt__()* member functions to overload) we can only have mixed modes (scalars and multivectors) if the first operand is a multivector.

Note: Except for < and > all the multivector operators have r-forms so that as long as one of the operands, left or right, is a multivector the other can be a multivector or a scalar (sympy symbol or integer).

Warning: Note that the operator order precedence is determined by python and is not necessarily that used by geometric algebra. It is **absolutely essential** to use parenthesis in multivector expressions containing ^, |, <, and/or >. As an example let *A* and *B* be any two multivectors. Then *A + A*B = A +(A*B)*, but *A+A^B = (2*A)^B* since in python the ^ operator has a lower precedence than the '+' operator. In geometric algebra the outer and inner products and the left and right contractions have a higher precedence than the geometric product and the geometric product has a higher precedence than addition and subtraction. In python the ^, |, <, and > all have a lower precedence than + and - while * has a higher precedence than + and -.

For those users who wish to define a default operator precedence the functions *define_precedence()* and *GAeval()* are available in the module *galgebra/precedence*.

`sympy.galgebra.precedence.`**`define_precedence`**(*gd*, *op_ord='<>|, ^, *'*)

Define the precedence of the multivector operations. The function *define_precedence()* must be called from the main program and the first argument *gd* must be set to *globals()*. The second argument *op_ord* determines the operator precedence for expressions input to the function *GAeval()*. The default value of *op_ord* is '<>|,^,*'. For the default value the <, >, and | operations have equal precedence followed by ^, and ^ is followed by *.

sympy.galgebra.precedence.**GAeval**(*s*, *pstr=False*)

The function *GAeval()* returns a multivector expression defined by the string *s* where the operations in the string are parsed according to the precedences defined by *define_precedence()*. *pstr* is a flag to print the input and output of *GAeval()* for debugging purposes. *GAeval()* works by adding parenthesis to the input string *s* with the precedence defined by *op_ord='<>|,^,*'*. Then the parsed string is converted to a sympy expression using the python *eval()* function. For example consider where *X*, *Y*, *Z*, and *W* are multivectors

```
define_precedence(globals())
V = GAeval('X|Y^Z*W')
```

The sympy variable *V* would evaluate to *((X|Y)^Z)*W*.

Vector Basis and Metric

The two structures that define the `MV` (multivector) class are the symbolic basis vectors and the symbolic metric. The symbolic basis vectors are input as a string with the symbol name separated by spaces. For example if we are calculating the geometric algebra of a system with three vectors that we wish to denote as *a0*, *a1*, and *a2* we would define the string variable:

```
basis = 'a0 a1 a2'
```

that would be input into the multivector setup function. The next step would be to define the symbolic metric for the geometric algebra of the basis we have defined. The default metric is the most general and is the matrix of the following symbols

$$g = \left[\begin{array}{ccc} (a0.a0) & (a0.a1) & (a0.a2) \\ (a0.a1) & (a1.a1) & (a1.a2) \\ (a0.a2) & (a1.a2) & (a2.a2) \end{array} \right] \tag{5.3}$$

where each of the g_{ij} is a symbol representing all of the dot products of the basis vectors. Note that the symbols are named so that $g_{ij} = g_{ji}$ since for the symbol function $(a0.a1) \neq (a1.a0)$.

Note that the strings shown in equation *5.3* (page 403) are only used when the values of g_{ij} are output (printed). In the *galgebra* module (library) the g_{ij} symbols are stored in a static member of the multivector class `MV` as the sympy matrix *MV.metric* (g_{ij} = *MV.metric[i,j]*).

The default definition of g can be overwritten by specifying a string that will define g. As an example consider a symbolic representation for conformal geometry. Define for a basis

```
basis = 'a0 a1 a2 n nbar'
```

and for a metric

```
metric = '# # # 0 0, # # # 0 0, # # # 0 0, 0 0 0 0 2, 0 0 0 2 0'
```

then calling *MV.setup(basis,metric)* would initialize the metric tensor

$$g = \left[\begin{array}{ccccc} (a0.a0) & (a0.a1) & (a0.a2) & 0 & 0 \\ (a0.a1) & (a1.a1) & (a1.a2) & 0 & 0 \\ (a0.a2) & (a1.a2) & (a2.a2) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{array} \right]$$

Here we have specified that *n* and *nbar* are orthonal to all the *a*'s, *(n.n) = (nbar.nbar) = 0*, and *(n.nbar) = 2*. Using # in the metric definition string just tells the program to use the default symbol for that value.

When *MV.setup* is called multivector representations of the basis local to the program are instantiated. For our first example that means that the symbolic vectors named *a0*, *a1*, and *a2* are created and returned from *MV.setup* via a tuple as in -

```
(a_1,a_2,a3) = MV.setup('a_1 a_2 a_3',metric=metric)
```

Note that the python variable name for a basis vector does not have to correspond to the name give in *MV.setup()*, one may wish to use a shorted python variable name to reduce programming (typing) errors, for example one could use -

```
(a1,a2,a3) = MV.setup('a_1 a_2 a_3',metric=metric)
```

or

```
(g1,g2,g3) = MV.setup('gamma_1 gamma_2 gamma_3',metric=metric)
```

so that if the latex printer is used *e1* would print as e_1 and *g1* as γ_1.

Note: Additionally *MV.setup* has simpified options for naming a set of basis vectors and for inputing an othogonal basis.

If one wishes to name the basis vectors e_x, e_y, and e_z then set *basis='e*x|y|z'* or to name γ_t, γ_x, γ_y, and γ_z then set *basis='gamma*t|x|y|z'*.

For the case of an othogonal basis if the signature of the vector space is $(1,1,1)$ (Euclidian 3-space) set *metric='[1,1,1]'* or if it is $(1,-1,-1,-1)$ (Minkowsi 4-space) set *metric='[1,-1,-1,-1]'*.

Representation and Reduction of Multivector Bases

In our symbolic geometric algebra all multivectors can be obtained from the symbolic basis vectors we have input, via the different operations available to geometric algebra. The first problem we have is representing the general multivector in terms terms of the basis vectors. To do this we form the ordered geometric products of the basis vectors and develop an internal representation of these products in terms of python classes. The ordered geometric products are all multivectors of the form $a_{i_1}a_{i_2}\dots a_{i_r}$ where $i_1 < i_2 < \cdots < i_r$ and $r \leq n$. We call these multivectors bases and represent them internally with noncommutative symbols so for example $a_1a_2a_3$ is represented by

```
Symbol('a_1*a_2*a_3',commutative=False)
```

In the simplist case of two basis vectors *a_1* and *a_2* we have a list of bases

```
MV.bases = [[Symbol('ONE',commutative=False)],[Symbol('a_1',commutative=False),\
            Symbol('a_2',commutative=False)],[Symbol('a_1*a_2',commutative=False)]]
```

Note: The reason that the base for the scalar component of the multivector is defined as *Symbol('ONE',commutative=False)*, a noncommutative symbol is because of the properties of the left and right contraction operators which are non commutative if one is contracting a multivector with a scalar.

For the case of the basis blades we have

```
MV.blades = [[Symbol('ONE',commutative=False)],[Symbol('a_1',commutative=False),\
            Symbol('a_2',commutative=False)],[Symbol('a_1^a_2',commutative=False)]]
```

Note: For all grades/pseudo-grades greater than one (vectors) the '*' in the name of the base symbol is replaced with a '^' in the name of the blade symbol so that for all basis bases and blades of grade/pseudo-grade greater than one there are different symbols for the corresponding bases and blades.

The function that builds all the required arrays and dictionaries upto the base multiplication table is shown below. *MV.dim* is the number of basis vectors and the *combinations* functions from *itertools* constructs the index tupels for the bases of each pseudo grade. Then the noncommutative symbol representing each base is constructed from each index tuple. *MV.ONE* is the noncommutative symbol for the scalar base. For example if *MV.dim = 3* then

```
MV.index = ((),((0,),(1,),(2,)),((0,1),(0,2),(1,2)),((0,1,2)))
```

Note: In the case that the metric tensor is diagonal (orthogonal basis vectors) both base and blade bases are identical and fewer arrays and dictionaries need to be constructed.

```
@staticmethod
def build_base_blade_arrays(debug):
    indexes = tuple(range(MV.dim))
    MV.index = [()]
    for i in indexes:
        MV.index.append(tuple(combinations(indexes,i+1)))
    MV.index = tuple(MV.index)

    #Set up base and blade and index arrays

    if not MV.is_orthogonal:
        MV.bases_flat = []
        MV.bases  = [MV.ONE]
        MV.base_to_index  = {MV.ONE:()}
        MV.index_to_base  = {():MV.ONE}
        MV.base_grades    = {MV.ONE:0}
        MV.base_grades[ONE] = 0

    MV.blades = [MV.ONE]
    MV.blades_flat = []
    MV.blade_grades    = {MV.ONE:0}
    MV.blade_grades[ONE] = 0
    MV.blade_to_index = {MV.ONE:()}
    MV.index_to_blade = {():MV.ONE}

    ig = 1 #pseudo grade and grade index
    for igrade in MV.index[1:]:
        if not MV.is_orthogonal:
            bases     = [] #base symbol array within pseudo grade
        blades    = [] #blade symbol array within grade
        ib = 0 #base index within grade
        for ibase in igrade:
            #build base name string
            (base_sym,base_str,blade_sym,blade_str) = MV.make_base_blade_symbol(ibase)

            if not MV.is_orthogonal:
                bases.append(base_sym)
                MV.bases_flat.append(base_sym)
```

```
            blades.append(blade_sym)
            MV.blades_flat.append(blade_sym)
            base_index = MV.index[ig][ib]

            #Add to dictionarys relating symbols and indexes
            if not MV.is_orthogonal:
                MV.base_to_index[base_sym]   = base_index
                MV.index_to_base[base_index] = base_sym
                MV.base_grades[base_sym]     = ig

            MV.blade_to_index[blade_sym] = base_index
            MV.index_to_blade[base_index] = blade_sym
            MV.blade_grades[blade_sym] = ig

            ib += 1
        ig += 1

        if not MV.is_orthogonal:
            MV.bases.append(tuple(bases))

        MV.blades.append(tuple(blades))

    if not MV.is_orthogonal:
        MV.bases       = tuple(MV.bases)
        MV.bases_flat  = tuple(MV.bases_flat)
        MV.bases_flat1 = (MV.ONE,)+MV.bases_flat
        MV.bases_set   = set(MV.bases_flat[MV.dim:])

    MV.blades       = tuple(MV.blades)
    MV.blades_flat  = tuple(MV.blades_flat)
    MV.blades_flat1 = (MV.ONE,)+MV.blades_flat
    MV.blades_set   = set(MV.blades_flat[MV.dim:])

    return
```

Base Representation of Multivectors

In terms of the bases defined as noncommutative sympy symbols the general multivector is a linear combination (scalar sympy coefficients) of bases so that for the case of two bases the most general multivector is given by -

```
A = A_0*MV.bases[0][0]+A__1*MV.bases[1][0]+A__2*MV.bases[1][1]+A__12*MV.bases[2][0]
```

If we have another multivector B to multiply with A we can calculate the product in terms of a linear combination of bases if we have a multiplication table for the bases.

Blade Representation of Multivectors

Since we can now calculate the symbolic geometric product of any two multivectors we can also calculate the blades corresponding to the product of the symbolic basis vectors using the formula

$$A_r \wedge b = \frac{1}{2}\left(A_r b - (-1)^r b A_r\right),$$

where A_r is a multivector of grade r and b is a vector. For our example basis the result is shown in Table 5.3 (page 406).

```
1 = 1
a0 = a0
a1 = a1
a2 = a2
a0^a1 = {-(a0.a1)}1+a0a1
a0^a2 = {-(a0.a2)}1+a0a2
a1^a2 = {-(a1.a2)}1+a1a2
a0^a1^a2 = {-(a1.a2)}a0+{(a0.a2)}a1+{-(a0.a1)}a2+a0a1a2
```

Table 5.3 (page 406). Bases blades in terms of bases.

The important thing to notice about Table 5.3 (page 406) is that it is a triagonal (lower triangular) system of equations so that using a simple back substitution algorithm we can solve for the pseudo bases in terms of the blades giving Table 5.4 (page 407).

```
1 = 1
a0 = a0
a1 = a1
a2 = a2
a0a1 = {(a0.a1)}1+a0^a1
a0a2 = {(a0.a2)}1+a0^a2
a1a2 = {(a1.a2)}1+a1^a2
a0a1a2 = {(a1.a2)}a0+{-(a0.a2)}a1+{(a0.a1)}a2+a0^a1^a2
```

Table 5.4 (page 407). Bases in terms of basis blades.

Using Table 5.4 (page 407) and simple substitution we can convert from a base multivector representation to a blade representation. Likewise, using Table 5.3 (page 406) we can convert from blades to bases.

Using the blade representation it becomes simple to program functions that will calculate the grade projection, reverse, even, and odd multivector functions.

Note that in the multivector class *MV* there is a class variable for each instantiation, *self.bladeflg*, that is set to *False* for a base representation and *True* for a blade representation. One needs to keep track of which representation is in use since various multivector operations require conversion from one representation to the other.

Warning: When the geometric product of two multivectors is calculated the module looks to see if either multivector is in blade representation. If either is the result of the geometric product is converted to a blade representation. One result of this is that if either of the multivectors is a simple vector (which is automatically a blade) the result will be in a blade representation. If *a* and *b* are vectors then the result *a*b* will be *(a.b)+a^b* or simply *a^b* if *(a.b) = 0*.

Outer and Inner Products, Left and Right Contractions

In geometric algebra any general multivector A can be decomposed into pure grade multivectors (a linear combination of blades of all the same order) so that in a n-dimensional vector space

$$A = \sum_{r=0}^{n} A_r$$

The geometric product of two pure grade multivectors A_r and B_s has the form

$$A_r B_s = \langle A_r B_s \rangle_{|r-s|} + \langle A_r B_s \rangle_{|r-s|+2} + \cdots + \langle A_r B_s \rangle_{r+s}$$

where $\langle\rangle_t$ projects the t grade components of the multivector argument. The inner and outer products of A_r and B_s are then defined to be

$$A_r \cdot B_s = \langle A_r B_s \rangle_{|r-s|}$$

$$A_r \wedge B_s = \langle A_r B_s \rangle_{r+s}$$

and

$$A \cdot B = \sum_{r,s} A_r \cdot B_s$$

$$A \wedge B = \sum_{r,s} A_r \wedge B_s$$

Likewise the right ($\lfloor$) and left ($\rfloor$) contractions are defined as

$$A_r \lfloor B_s = \left\{ \begin{array}{cc} \langle A_r B_s \rangle_{r-s} & r \geq s \\ 0 & r < s \end{array} \right\}$$

$$A_r \rfloor B_s = \left\{ \begin{array}{cc} \langle A_r B_s \rangle_{s-r} & s \geq r \\ 0 & s < r \end{array} \right\}$$

and

$$A \lfloor B = \sum_{r,s} A_r \lfloor B_s$$

$$A \rfloor B = \sum_{r,s} A_r \rfloor B_s$$

Warning: In the *MV* class we have overloaded the ^ operator to represent the outer product so that instead of calling the outer product function we can write *mv1^ mv2*. Due to the precedence rules for python it is **absolutely essential** to enclose outer products in parenthesis.

Warning: In the *MV* class we have overloaded the | operator for the inner product, > operator for the right contraction, and < operator for the left contraction. Instead of calling the inner product function we can write *mv1|mv2*, *mv1>mv2*, or *mv1<mv2* respectively for the inner product, right contraction, or left contraction. Again, due to the precedence rules for python it is **absolutely essential** to enclose inner products and/or contractions in parenthesis.

Reverse of Multivector

If A is the geometric product of r vectors

$$A = a_1 \ldots a_r$$

then the reverse of A designated $A^\dagger$ is defined by

$$A^\dagger \equiv a_r \ldots a_1.$$

The reverse is simply the product with the order of terms reversed. The reverse of a sum of products is defined as the sum of the reverses so that for a general multivector A we have

$$A^\dagger = \sum_{i=0}^{N} \langle A \rangle_i^\dagger$$

but

$$\langle A\rangle_i{}^\dagger = (-1)^{\frac{i(i-1)}{2}}\langle A\rangle_i \tag{5.4}$$

which is proved by expanding the blade bases in terms of orthogonal vectors and showing that equation *5.4* (page 409) holds for the geometric product of orthogonal vectors.

The reverse is important in the theory of rotations in n-dimensions. If R is the product of an even number of vectors and $RR^\dagger = 1$ then $RaR^\dagger$ is a composition of rotations of the vector a. If R is the product of two vectors then the plane that R defines is the plane of the rotation. That is to say that $RaR^\dagger$ rotates the component of a that is projected into the plane defined by a and b where $R = ab$. R may be written $R = e^{\frac{\theta}{2}U}$, where θ is the angle of rotation and u is a unit blade $\left(u^2 = \pm 1\right)$ that defines the plane of rotation.

Reciprocal Frames

If we have M linearly independent vectors (a frame), $a_1, \ldots, a_M$, then the reciprocal frame is $a^1, \ldots, a^M$ where $a_i \cdot a^j = \delta_i^j$, δ_i^j is the Kronecker delta (zero if $i \neq j$ and one if $i = j$). The reciprocal frame is constructed as follows:

$$E_M = a_1 \wedge \cdots \wedge a_M$$

$$E_M^{-1} = \frac{E_M}{E_M^2}$$

Then

$$a^i = (-1)^{i-1}\left(a_1 \wedge \cdots \wedge \breve{a}_i \wedge \cdots \wedge a_M\right) E_M^{-1}$$

where $\breve{a}_i$ indicates that a_i is to be deleted from the product. In the standard notation if a vector is denoted with a subscript the reciprocal vector is denoted with a superscript. The multivector setup function *MV.setup(basis,metric,rframe)* has the argument *rframe* with a default value of *False*. If it is set to *True* the reciprocal frame of the basis vectors is calculated. Additionally there is the function *reciprocal_frame(vlst,names='')* external to the *MV* class that will calculate the reciprocal frame of a list, *vlst*, of vectors. If the argument *names* is set to a space delimited string of names for the vectors the reciprocal vectors will be given these names.

Geometric Derivative

If F is a multivector field that is a function of a vector $x = x^i \boldsymbol{e}_i$ (we are using the summation convention that pairs of subscripts and superscripts are summed over the dimension of the vector space) then the geometric derivative ∇F is given by (in this section the summation convention is used):

$$\nabla F = \boldsymbol{e}^i \frac{\partial F}{\partial x^i}$$

If F_R is a grade-R multivector and $F_R = F_R^{i_1 \ldots i_R} \boldsymbol{e}_{i_1} \wedge \cdots \wedge \boldsymbol{e}_{i_R}$ then

$$\nabla F_R = \frac{\partial F_R^{i_1 \ldots i_R}}{\partial x^j} \boldsymbol{e}^j \left(\boldsymbol{e}_{i_1} \wedge \cdots \wedge \boldsymbol{e}_{i_R}\right)$$

Note that $\boldsymbol{e}^j \left(\boldsymbol{e}_{i_1} \wedge \cdots \wedge \boldsymbol{e}_{i_R}\right)$ can only contain grades $R-1$ and $R+1$ so that ∇F_R also can only contain those grades. For a grade-R multivector F_R the inner (div) and outer (curl) derivatives are defined as

$$\nabla \cdot F_R = \langle \nabla F_R \rangle_{R-1}$$

and

$$\nabla \wedge F_R = \langle \nabla F_R \rangle_{R+1}$$

For a general multivector function F the inner and outer derivatives are just the sum of the inner and outer dervatives of each grade of the multivector function.

Curvilinear coordinates are derived from a vector function $x(\boldsymbol{\theta})$ where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_N)$ where the number of coordinates is equal to the dimension of the vector space. In the case of 3-dimensional spherical coordinates $\boldsymbol{\theta} = (r, \theta, \phi)$ and the coordinate generating function $x(\boldsymbol{\theta})$ is

$$x = r\cos(\phi)\sin(\theta)\,\boldsymbol{e_x} + r\sin(\phi)\sin(\theta)\,\boldsymbol{e_y} + r\cos(\theta)\,\boldsymbol{e_z}$$

A coordinate frame is derived from x by $\boldsymbol{e}_i = \dfrac{\partial x}{\partial \theta^i}$. The following show the frame for spherical coordinates.

$$\boldsymbol{e}_r = \cos(\phi)\sin(\theta)\,\boldsymbol{e_x} + \sin(\phi)\sin(\theta)\,\boldsymbol{e_y} + \cos(\theta)\,\boldsymbol{e_z}$$
$$\boldsymbol{e}_\theta = \cos(\phi)\cos(\theta)\,\boldsymbol{e_x} + r\cos(\theta)\sin(\phi)\,\boldsymbol{e_y} - r\sin(\theta)\,\boldsymbol{e_z}$$
$$\boldsymbol{e}_\phi = -r\sin(\phi)\sin(\theta)\,\boldsymbol{e_x} + r\cos(\phi)\sin(\theta)\,\boldsymbol{e_y}$$

The coordinate frame generated in this manner is not necessarily normalized so define a normalized frame by

$$\hat{\boldsymbol{e}}_i = \frac{\boldsymbol{e}_i}{\sqrt{|\boldsymbol{e}_i^2|}} = \frac{\boldsymbol{e}_i}{|\boldsymbol{e}_i|}$$

This works for all $\boldsymbol{e}_i^2 \neq 0$ since we have defined $|\boldsymbol{e}_i| = \sqrt{|\boldsymbol{e}_i^2|}$. For spherical coordinates the normalized frame vectors are

$$\hat{\boldsymbol{e}}_r = \cos(\phi)\sin(\theta)\,\boldsymbol{e_x} + \sin(\phi)\sin(\theta)\,\boldsymbol{e_y} + \cos(\theta)\,\boldsymbol{e_z}$$
$$\hat{\boldsymbol{e}}_\theta = \cos(\phi)\cos(\theta)\,\boldsymbol{e_x} + \cos(\theta)\sin(\phi)\,\boldsymbol{e_y} - \sin(\theta)\,\boldsymbol{e_z}$$
$$\hat{\boldsymbol{e}}_\phi = -\sin(\phi)\,\boldsymbol{e_x} + \cos(\phi)\,\boldsymbol{e_y}$$

The geometric derivative in curvilinear coordinates is given by

$$\begin{aligned}
\nabla F_R &= \boldsymbol{e}^i \frac{\partial}{\partial x^i}\left(F_R^{i_1\ldots i_R}\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right)\\
&= \boldsymbol{e^j}\frac{\partial}{\partial \theta^j}\left(F_R^{i_1\ldots i_R}\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right)\\
&= \left(\frac{\partial}{\partial\theta^j}F_R^{i_1\ldots i_R}\right)\boldsymbol{e^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right) + F_R^{i_1\ldots i_R}\boldsymbol{e^j}\frac{\partial}{\partial\theta^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right)\\
&= \left(\frac{\partial}{\partial\theta^j}F_R^{i_1\ldots i_R}\right)\boldsymbol{e^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right) + F_R^{i_1\ldots i_R}C\left\{\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right\}
\end{aligned}$$

where

$$C\left\{\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right\} = \boldsymbol{e^j}\frac{\partial}{\partial\theta^j}\left(\hat{\boldsymbol{e}}_{i_1}\wedge\cdots\wedge\hat{\boldsymbol{e}}_{i_R}\right)$$

are the connection multivectors for the curvilinear coordinate system. For a spherical coordinate system they are

$$C\left\{\hat{\boldsymbol{e}}_r\right\} = \frac{2}{r}$$
$$C\left\{\hat{\boldsymbol{e}}_\theta\right\} = \frac{\cos(\theta)}{r\sin(\theta)} + \frac{1}{r}\hat{\boldsymbol{e}}_r\wedge\hat{\boldsymbol{e}}_\theta$$
$$C\left\{\hat{\boldsymbol{e}}_\phi\right\} = \frac{1}{r}\hat{\boldsymbol{e}}_{\boldsymbol{r}}\wedge\hat{\boldsymbol{e}}_\phi + \frac{\cos(\theta)}{r\sin(\theta)}\hat{\boldsymbol{e}}_\theta\wedge\hat{\boldsymbol{e}}_\phi$$
$$C\left\{\hat{e}_r\wedge\hat{e}_\theta\right\} = -\frac{\cos(\theta)}{r\sin(\theta)}\hat{\boldsymbol{e}}_r + \frac{1}{r}\hat{\boldsymbol{e}}_\theta$$
$$C\left\{\hat{\boldsymbol{e}}_r\wedge\hat{\boldsymbol{e}}_\phi\right\} = \frac{1}{r}\hat{\boldsymbol{e}}_\phi - \frac{\cos(\theta)}{r\sin(\theta)}\hat{\boldsymbol{e}}_r\wedge\hat{\boldsymbol{e}}_\theta\wedge\hat{\boldsymbol{e}}_\phi$$
$$C\left\{\hat{\boldsymbol{e}}_\theta\wedge\hat{\boldsymbol{e}}_\phi\right\} = \frac{2}{r}\hat{\boldsymbol{e}}_r\wedge\hat{\boldsymbol{e}}_\theta\wedge\hat{\boldsymbol{e}}_\phi$$
$$C\left\{\hat{\boldsymbol{e}}_r\wedge\hat{\boldsymbol{e}}_\theta\wedge\hat{\boldsymbol{e}}_\phi\right\} = 0$$

Numpy, LaTeX, and Ansicon Installation

To install the geometric algebra module on windows,linux, or OSX perform the following operations

1. Install sympy. *galgebra* is included in sympy.
2. To install texlive in linux or windows
   (a) Go to <http://www.tug.org/texlive/acquire-netinstall.html> and click on "install-tl.zip" o download
   (b) Unzip "install-tl.zip" anywhere on your machine
   (c) Open the file "readme.en.html" in the "readme-html.dir" directory. This file contains the information needed to install texlive.
   (d) Open a terminal (console) in the "install-tl-XXXXXX" directory
   (e) Follow the instructions in "readme.en.html" file to run the install-tl.bat file in windows or the install-tl script file in linux.
3. For OSX install mactex from <http://tug.org/mactex/>.
4. Install python-nympy if you want to calculate numerical matrix functons (determinant, inverse, eigenvalues, etc.). For windows go to <http://sourceforge.net/projects/numpy/files/NumPy/1.6.2/> and install the distribution of numpy appropriate for your system. For OSX go to <http://sourceforge.net/projects/numpy/files/NumPy/1.6.1/>.
5. It is strongly suggested that you go to <http://www.geany.org/Download/Releases> and install the version of the "geany" editor appropriate for your system.
6. If you wish to use "enhance_print" on windows -
   (a) Go to <https://github.com/adoxa/ansicon/downloads> and download "ansicon"
   (b) In the Edit -> Preferences -> Tools menu of "geany" enter into the Terminal input the full path of "ansicon.exe"

After installation if you are doing you code development in the *galgebra* directory you need only include

```
from sympy.galgebra.printing import xdvi,enhance_print
from sympy.galgebra.ga import *
```

to use the *galgebra* module.

In addition to the code shown in the examples section of this document there are more examples in the Examples directory under the *galgebra* directory.

Module Components

Initializing Multivector Class

The multivector class is initialized with:

`sympy.galgebra.ga.MV.`**`setup`**(*basis, metric=None, coords=None, rframe=False, debug=False, curv=(None, None)*)

The *basis* and *metric* parameters were described in section *Vector Basis and Metric* (page 403). If *rframe=True* the reciprocal frame of the symbolic bases vectors is calculated. If *debug=True* the data structure required to initialize the `MV` class are printer out. *coords* is a tuple of `sympy` (page 612) symbols equal in length to the number of basis

vectors. These symbols are used as the arguments of a multivector field as a function of position and for calculating the derivatives of a multivector field (if *coords* is defined then *rframe* is automatically set equal to *True*). Additionally, `MV.setup()` calculates the pseudo scalar, I and makes them available to the programmer as *MV.I* and *MV.Iinv*.

`MV.setup()` always returns a tuple containing the basis vectors (as multivectors) so that if we have the code

```
(e1,e2,e3) = MV.setup('e_1 e_2 e_3')
```

then we can define a multivector by the expression

```
(a1,a2,a3) = symbols('a__1 a__2 a__3')
A = a1*e1+a2*e2+a3*e3
```

Another option is

```
(e1,e2,e3) = MV.setup('e*1|2|3')
```

which produce the same results as the previous method. Note that if we had used

```
(e1,e2,e3) = MV.setup('e*x|y|z')
```

then the basis vectors would have been labeled *e_x*, *e_y*, and *e_z*. If *coords* is defined then `MV.setup()` returns the tuple

```
X = (x,y.z) = symbols('x y z')
(ex,ey,ez,grad) = MV.setup('e',coords=X)
```

the basis vectros are again labeled *e_x*, *e_y*, and *e_z* and the additional vector *grad* is returned. *grad* acts as the gradient operator (geometric derivative) so that if `F()` is a multivector function of *(x,y,z)* then

```
DFl = grad*F
DFr = F*grad
```

are the left and right geometric derivatives of `F()`.

The final parameter in `MV.setup()` is *curv* which defines a curvilinear coordinate system. If 3-dimensional spherical coordinates are required we would define -

```
X = (r,th,phi) = symbols('r theta phi')
curv = [[r*cos(phi)*sin(th),r*sin(phi)*sin(th),r*cos(th)],[1,r,r*sin(th)]]
(er,eth,ephi,grad) = MV.setup('e_r e_theta e_phi',metric='[1,1,1]',coords=X,curv=curv)
```

The first component of *curv* is

```
[r*cos(phi)*sin(th),r*sin(phi)*sin(th),r*cos(th)]
```

This is the position vector for the spherical coordinate system expressed in terms of the rectangular coordinate components given in terms of the spherical coordinates *r*, *th*, and *phi*. The second component of *curv* is

```
[1,r,r*sin(th)]
```

The components of *curv[1]* are the normalizing factors for the basis vectors of the spherical coordinate system that are calculated from the derivatives of *curv[0]* with respect to the coordinates *r*, *th*, and *phi*. In theory the normalizing factors can be calculated from the derivatives of *curv[0]*. In practice one cannot currently specify in sympy that the square of a function is always positive which leads to problems when the normalizing

factor is the square root of a squared function. To avoid these problems the normalizing factors are explicitly defined in *curv[1]*.

Note: In the case of curvlinear coordinates *debug* also prints the connection multivectors.

Instantiating a Multivector

Now that grades and bases have been described we can show all the ways that a multivector can be instantiated. As an example assume that the multivector space is initialized with

```
(e1,e2,e3) = MV.setup('e_1 e_2 e_3')
```

then multivectors could be instantiated with

```
(a1,a2,a3) = symbols('a__1 a__2 a__3')
x = a1*e1+a2*e2+a3*e3
y = x*e1*e2
z = x|y
w = x^y
```

or with the multivector class constructor:

class `sympy.galgebra.ga.MV`(*base=None, mvtype=None, fct=False, blade_rep=True*)

base is a string that defines the name of the multivector for output purposes. *base* and *mvtype* are defined by the following table and *fct* is a switch that will convert the symbolic coefficients of a multivector to functions if coordinate variables have been defined when `MV.setup()` is called:

mvtype	base	result
default	default	Zero multivector
scalar	string s	symbolic scalar of value Symbol(s)
vector	string s	symbolic vector
grade2 or bivector	string s	symbolic bivector
grade	string s,n	symbolic n-grade multivector
pseudo	string s	symbolic pseudoscalar
spinor	string s	symbolic even multivector
mv	string s	symbolic general multivector
default	sympy scalar c	zero grade multivector with coefficient c
default	multivector	copy constructor for multivector

If the *base* argument is a string s then the coefficients of the resulting multivector are named as follows:

The grade r coefficients consist of the base string, s, followed by a double underscore, __, and an index string of r symbols. If *coords* is defined the index string will consist of coordinate names in a normal order defined by the *coords* tuple. If *coords* is not defined the index string will be integers in normal (ascending) order (for an n dimensional vector space the indices will be 1 to n). The double underscore is used because the latex printer interprets it as a superscript and superscripts in the coefficients will balance subscripts in the bases.

For example if If *coords=(x,y,z)* and the base is *A*, the list of all possible coefficients for the most general multivector would be *A*, *A__x*, *A__y*, *A__z*, *A__xy*, *A__xz*, *A__yz*, and *A_xyz*. If the latex printer is used and *e* is the base for the

basis vectors then the pseudo scalar would print as $A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}$. If coordinates are not defined it would print as $A^{123}\boldsymbol{e_1}\wedge\boldsymbol{e_2}\wedge\boldsymbol{e_3}$. For printed output all multivectors are represented in terms of products of the basis vectors, either as geometric products or wedge products. This is also true for the output of expressions containing reciprocal basis vectors.

If the *fct* argument of `MV()` is set to *True* and the *coords* argument in `MV.setup()` is defined the symbolic coefficients of the multivector are functions of the coordinates.

Basic Multivector Class Functions

`sympy.galgebra.ga.MV.`**`convert_to_blades`**(*self*)
Convert multivector from the base representation to the blade representation. If multivector is already in blade representation nothing is done.

`sympy.galgebra.ga.MV.`**`convert_from_blades`**(*self*)
Convert multivector from the blade representation to the base representation. If multivector is already in base representation nothing is done.

`sympy.galgebra.ga.MV.`**`dd`**(*self*, *v*)
For a mutivector function *F* and a vector *v* then *F.dd(v)* is the directional derivate of *F* in the direction *v*, $(v\cdot\nabla)F$.

`sympy.galgebra.ga.MV.`**`diff`**(*self*, *var*)
Calculate derivative of each multivector coefficient with resepect to variable *var* and form new multivector from coefficients.

`sympy.galgebra.ga.MV.`**`dual`**(*self*)
Return dual of multivector which is multivector left multiplied by pseudoscalar *MV.I* ([Hestenes] (page 1624),p22).

`sympy.galgebra.ga.MV.`**`even`**(*self*)
Return the even grade components of the multivector.

`sympy.galgebra.ga.MV.`**`exp`**(*self*, *alpha=1*, *norm=0*, *mode='T'*)
Return exponential of a blade (if self is not a blade error message is generated). If A is the blade then $e^{\alpha A}$ is returned where the default *mode*, *'T'*, assumes $AA<0$ so that

$$e^{\alpha A}=\cos\left(\alpha\sqrt{-A^2}\right)+\sin\left(\alpha\sqrt{-A^2}\right)\frac{A}{\sqrt{-A^2}}.$$

If the mode is not *'T'* then $AA>0$ is assumed so that

$$e^{\alpha A}=\cosh\left(\alpha\sqrt{A^2}\right)+\sinh\left(\alpha\sqrt{A^2}\right)\frac{A}{\sqrt{A^2}}.$$

If $norm=N>0$ then

$$e^{\alpha A}=\cos\left(\alpha N\right)+\sin\left(\alpha N\right)\frac{A}{N}$$

or

$$e^{\alpha A}=\cosh\left(\alpha N\right)+\sinh\left(\alpha N\right)\frac{A}{N}$$

depending on the value of *mode*.

`sympy.galgebra.ga.MV.`**`expand`**(*self*)
Return multivector in which each coefficient has been expanded using sympy *expand()* function.

`sympy.galgebra.ga.MV.`**`factor`**(*self*)
Apply the *sympy factor* function to each coefficient of the multivector.

sympy.galgebra.ga.MV.**func**(*self, fct*)
Apply the *sympy* scalar function *fct* to each coefficient of the multivector.

sympy.galgebra.ga.MV.**grade**(*self, igrade=0*)
Return a multivector that consists of the part of the multivector of grade equal to *igrade*. If the multivector has no *igrade* part return a zero multivector.

sympy.galgebra.ga.MV.**inv**(*self*)
Return the inverse of the multivector if *self*sefl.rev()* is a nonzero ctor.

sympy.galgebra.ga.MV.**norm**(*self*)
Return the norm of the multvector M (*M.norm()*) defined by $\sqrt{MM^\dagger}$. If $MM^\dagger$ is a scalar (a sympy scalar is returned). If $MM^\dagger$ in not a scalar the program exits with an error message.

sympy.galgebra.ga.MV.**norm**(*self*)
Return the square of norm of the multvector M (*M.norm2()*) defined by $MM^\dagger$. If $MM^\dagger$ is a scalar (a sympy scalar is returned). If $MM^\dagger$ in not a scalar the program exits with an error message.

sympy.galgebra.ga.MV.**scalar**(*self*)
Return the coefficient (sympy scalar) of the scalar part of a multivector.

sympy.galgebra.ga.MV.**simplify**(*self*)
Return multivector where sympy simplify function has been applied to each coefficient of the multivector.

sympy.galgebra.ga.MV.**subs**(*self, x*)
Return multivector where sympy subs function has been applied to each coefficient of multivector for argument dictionary/list x.

sympy.galgebra.ga.MV.**rev**(*self*)
Return the reverse of the multivector. See section *Reverse of Multivector* (page 408).

sympy.galgebra.ga.MV.**set_coef**(*self, grade, base, value*)
Set the multivector coefficient of index *(grade,base)* to *value*.

sympy.galgebra.ga.MV.**trigsimp**(*self, **kwargs*)
Apply the *sympy* trignometric simplification fuction *trigsimp* to each coefficient of the multivector. ***kwargs* are the arguments of trigsimp. See *sympy* documentation on *trigsimp* for more information.

Basic Multivector Functions

sympy.galgebra.ga.**Com**(A, B)
Calulate commutator of multivectors A and B. Returns $(AB - BA)/2$.

sympy.galgebra.ga.**DD**(*v, f*)
Calculate directional derivative of multivector function *f* in direction of vector *v*. Returns *f.dd(v)*.

sympy.galgebra.ga.**Format**(*Fmode=True, Dmode=True, ipy=False*)
See latex printing.

sympy.galgebra.precedence.**GAeval**(*s, pstr=False*)
Returns multivector expression for string *s* with operator precedence for string *s* defined by inputs to function *define_precedence()*. if *pstr=True s* and *s* with parenthesis added to enforce operator precedence are printed.

sympy.galgebra.ga.**Nga**(*x*, *prec=5*)
If *x* is a multivector with coefficients that contain floating point numbers, *Nga()* rounds all these numbers to a precision of *prec* and returns the rounded multivector.

sympy.galgebra.ga.**ReciprocalFrame**(*basis*, *mode='norm'*)
If *basis* is a list/tuple of vectors, *ReciprocalFrame()* returns a tuple of reciprocal vectors. If *mode=norm* the vectors are normalized. If *mode* is anything other than *norm* the vectors are unnormalized and the normalization coefficient is added to the end of the tuple. One must divide by the coefficient to normalize the vectors.

sympy.galgebra.ga.**ScalarFunction**(*TheFunction*)
If *TheFuction* is a real *sympy* fuction a scalar multivector function is returned.

sympy.galgebra.ga.**cross**(*M1*, *M2*)
If *M1* and *M2* are 3-dimensional euclidian vectors the vector cross product is returned, $v_1 \times v_2 = -I\,(v_1 \wedge v_2)$.

sympy.galgebra.precedence.**define_precedence**(*gd*, *op_ord='<>|, ^, *'*)
This is used with the *GAeval()* fuction to evaluate a string representing a multivector expression with a revised operator precedence. *define_precedence()* redefines the operator precedence for multivectors. *define_precedence()* must be called in the main program an the argument *gd* must be *globals()*. The argument *op_ord* defines the order of operator precedence from high to low with groups of equal precedence separated by commas. the default precedence *op_ord='<>|,^,*'* is that used by Hestenes ([Hestenes] (page 1624),p7,[Doran]_,p38).

sympy.galgebra.ga.**dual**(*M*)
Return the dual of the multivector *M*, MI^{-1}.

sympy.galgebra.ga.**inv**(*B*)
If for the multivector B, $BB^{\dagger}$ is a nonzero scalar, return $B^{-1} = B^{\dagger}/(BB^{\dagger})$.

sympy.galgebra.ga.**proj**(*B*, *A*)
Project blade A on blade B returning $(A \lfloor B)\,B^{-1}$.

sympy.galgebra.ga.**refl**(*B*, *A*)
Reflect blade *A* in blade *B*. If *r* is grade of *A* and *s* is grade of *B* returns $(-1)^{s(r+1)}BAB^{-1}$.

sympy.galgebra.ga.**rot**(*itheta*, *A*)
Rotate blade *A* by 2-blade *itheta*. Is is assumed that *itheta*itheta > 0* so that the rotation is Euclidian and not hyperbolic so that the angle of rotation is *theta = itheta.norm()*. Ther in 3-dimensional Euclidian space. *theta* is the angle of rotation (scalar in radians) and *n* is the vector axis of rotation. Returned is the rotor *cos(theta)+sin(theta)*N* where *N* is the normalized dual of *n*.

Multivector Derivatives

The various derivatives of a multivector function is accomplished by multiplying the gradient operator vector with the function. The gradiant operation vector is returned by the *MV.setup()* function if coordinates are defined. For example if we have for a 3-D vector space

```
X = (x,y,z) = symbols('x y z')
(ex,ey,ez,grad) = MV.setup('e*x|y|z',metric='[1,1,1]',coords=X)
```

Then the gradient operator vector is *grad* (actually the user can give it any name he wants to). Then the derivatives of the multivector function *F* are given by

```
F = MV('F','mv',fct=True)
```

$$\nabla F = grad * F$$
$$F\nabla = F * grad$$
$$\nabla \wedge F = grad \wedge F$$
$$F \wedge \nabla = F \wedge grad$$
$$\nabla \cdot F = grad|F$$
$$F \cdot \nabla F = F|grad$$
$$\nabla \lfloor F = grad < F$$
$$F \lfloor \nabla = F < grad$$
$$\nabla \rfloor F = grad > F$$
$$F \rfloor \nabla = F > grad$$

The preceding code block gives examples of all possible multivector derivatives of the multivector function *F* where * give the left and right geometric derivatives, ^ gives the left and right exterior (curl) derivatives, | gives the left and right interior (div) derivatives, < give the left and right derivatives for the left contraction, and > give the left and right derivatives for the right contraction. To understand the left and right derivatives see a reference on geometric calculus ([Doran] (page 1624),chapter6).

If one is taking the derivative of a complex expression that expression should be in parenthesis. Additionally, whether or not one is taking the derivative of a complex expression the *grad* vector and the expression it is operating on should always be in parenthesis unless the grad operator and the expression it is operating on are the only objects in the expression.

Vector Manifolds

In addtition to the *galgebra* module there is a *manifold* module that allows for the definition of a geometric algebra and calculus on a vector manifold. The vector mainfold is defined by a vector function of some coordinates in an embedding vector space ([Doran] (page 1624),p202,[Hestenes]_,p139). For example the unit 2-sphere would be the collection of vectors on the unit shpere in 3-dimensions with possible coordinates of θ and ϕ the angles of elevation and azimuth. A vector function $X(\theta, \phi)$ that defines the manifold would be given by

$$X(\theta, \phi) = \cos(\theta)\, \boldsymbol{e_z} + \cos(\theta)\left(\cos(\phi)\, \boldsymbol{e_x} + \sin(\phi)\, \boldsymbol{e_y}\right)$$

The module *manifold.py* is transitionary in that all calculation are performed in the embedding vector space (geometric algebra). Thus due to the limitations on *sympy*'s *simplify()* and *trigsimp()*, simple expressions may appear to be very complicated since they are expressed in terms of the basis vectors (bases/blades) of the embedding space and not in terms of the vector space (geometric algebra) formed from the manifold's basis vectors. A future implementation of *Manifold.py* will correct this difficiency. The member functions of the vector manifold follow.

`sympy.galgebra.ga.`**`Manifold`**(*x*, *coords*, *debug=False*, *I=None*)

Initializer for vector manifold where *x* is the vector function of the *coords* that defines the manifold and *coords* is the list/tuple of sympy symbols that are the coordinates. The basis vectors of the manifold as a fuction of the coordinates are returned as a tuple. *I* is the pseudo scalar for the manifold. The default is for the initializer to calculate *I*, however for complicated *x* functions (especially where trigonometric functions of the coordinates are involved) it is sometimes a good idea to calculate *I* separately and input it to *Manifold()*.

`sympy.galgebra.ga.`**`Basis`**(*self*)

Return the basis vectors of the manifold as a tuple.

`sympy.galgebra.ga.`**`DD`**(*self*, *v*, *F*, *opstr=False*)

Return the manifold directional derivative of a multivector function F defined on the manifold in the vector direction v.

`sympy.galgebra.ga.`**`Grad`**(*self*, *F*)

Return the manifold multivector derivative of the multivector function F defined on the manifold.

`sympy.galgebra.ga.`**`Proj`**(*self*, *F*)

Return the projection of the multivector F onto the manifold tangent space.

An example of a simple vector manifold is shown below which demonstrates the instanciation of a manifold, the defining of vector and scalar functions on the manifold and the calculation of the geometric derivative of those functions.

```
def Simple_manifold_with_vector_function_derivative():
    Print_Function()
    coords = (x,y,z) = symbols('x y z')
    basis = (ex, ey, ez, grad) = \
            MV.setup('e_x e_y e_z',metric='[1,1,1]',coords=coords)
    # Define surface
    mfvar = (u,v) = symbols('u v')
    X = u*ex+v*ey+(u**2+v**2)*ez
    print '\\f{X}{u,v} =',X
    MF = Manifold(X,mfvar)
    (eu,ev) = MF.Basis()
    # Define field on the surface.
    g = (v+1)*log(u)
    print '\\mbox{Scalar Function: } g =',g
    dg = MF.grad*g
    dg.Fmt(3,'\\mbox{Scalar Function Derivative: } \\nabla g')
    print '\\eval{\\nabla g}{(1,0)} =',dg.subs({u:1,v:0})
    # Define vector field on the surface
    G = v**2*eu+u**2*ev
    print '\\mbox{Vector Function: } G =',G
    dG = MF.grad*G
    dG.Fmt(3,'\\mbox{Vector Function Derivative: } \\nabla G')
    print '\\eval{\\nabla G}{(1,0)} =',dG.subs({u:1,v:0})
    return
```

Code Output:

$$X(u,v) = u\boldsymbol{e_x} + v\boldsymbol{e_y} + \left(u^2+v^2\right)\boldsymbol{e_z}$$

Scalar Function: $g = (v+1)\log(u)$

$$\begin{aligned}\text{Scalar Function Derivative: } \nabla g = &\left(\frac{-4u^2v\log(u)+4v^3+4v^2+v+1}{u\left(4u^2+4v^2+1\right)}\right)\boldsymbol{e_x}\\ &+\left(\frac{4u^2\log(u)-4v^2-4v+\log(u)}{4u^2+4v^2+1}\right)\boldsymbol{e_y}\\ &+\left(2\frac{v\log(u)+v+1}{4u^2+4v^2+1}\right)\boldsymbol{e_z}\end{aligned}$$

$$\nabla g|_{(1,0)} = \frac{1}{5}\boldsymbol{e_x} + \frac{2}{5}\boldsymbol{e_z}$$

Vector Function: $G = v^2\boldsymbol{e_x} + u^2\boldsymbol{e_y} + (2uv(u+v))\boldsymbol{e_z}$

$$\begin{aligned}\text{Vector Function Derivative: } \nabla G = &4\frac{uv(u+v)}{4u^2+4v^2+1}\\ &+\left(2\frac{-4u^2v+4uv^2+u-v}{4u^2+4v^2+1}\right)\boldsymbol{e_x}\wedge\boldsymbol{e_y}\\ &+\left(2\frac{v\left(-4u^3-8u^2v+8uv^2+2u+4v^3-v\right)}{4u^2+4v^2+1}\right)\boldsymbol{e_x}\wedge\boldsymbol{e_z}\\ &+\left(2\frac{u\left(4u^3+8u^2v-8uv^2-u-4v^3+2v\right)}{4u^2+4v^2+1}\right)\boldsymbol{e_y}\wedge\boldsymbol{e_z}\end{aligned}$$

$$\nabla G|_{(1,0)} = \frac{2}{5}\boldsymbol{e_x}\wedge\boldsymbol{e_y} + \frac{6}{5}\boldsymbol{e_y}\wedge\boldsymbol{e_z}$$

Standard Printing

Printing of multivectors is handled by the module *galgebra/printing* which contains a string printer class, *GA_Printer* derived from the sympy string printer class and a latex printer class, *GA_LatexPrinter*, derived from the sympy latex printer class. Additionally, there is an *enhanced_print* class that enhances the console output of sympy to make the printed output multivectors, functions, and derivatives more readable. *enhanced_print* requires an ansi console such as is supplied in linux or the program *ansicon* (github.com/adoxa/ansicon) for windows which replaces *cmd.exe*.

For a windows user the simplest way to implement ansicon is to use the *geany* editor and in the Edit->Preferences->Tools menu replace *cmd.exe* with *ansicon.exe* (be sure to supply the path to *ansicon*).

If *enhanced_print* is called in a program (linux) when multivectors are printed the basis blades or bases are printed in bold text, functions are printed in red, and derivative operators in green.

For formatting the multivector output there is the member function

`sympy.galgebra.ga.`**`Fmt`**(*self, fmt=1, title=None*)

Fmt is used to control how the multivector is printed with the argument *fmt*. If *fmt=1* the entire multivector is printed on one line. If *fmt=2* each grade of the multivector is printed on one line. If *fmt=3* each component (base) of the multivector is printed on one line. If a *title* is given then *title = multivector* is printed. If the usual print command is used the entire multivector is printed on one line.

`sympy.galgebra.ga.`**`ga_print_on`**()

Redirects printer output from standard *sympy* print handler. Needed if one wishes to use compact forms of *function* and *derivative* output strings for interactive use.

`sympy.galgebra.ga.`**`ga_print_off`**()

Restores standard *sympy* print handler.

`sympy.galgebra.printing.`**`GA_Printer`**()

Context handler for use inside Sympy code. The code indented under the *with GA_Printer():* statement will run with the compact forms of *function* and *derivative* output strings, and have the printing restored to standard *sympy* printing after it has finished, even if it is aborted with an exception.

Latex Printing

For latex printing one uses one functions from the *galgebra* module and one function from the *galgebra/printing* module. The functions are

`sympy.galgebra.printing.`**`Format`**(*Fmode=True, Dmode=True, ipy=False*)

This function from the *galgebra* module turns on latex printing with the following options

argument	value	result
Fmode	*True*	Print functions without argument list, f
	False	Print functions with standard sympy latex formatting, $f(x,y,z)$
Dmode	*True*	Print partial derivatives with condensed notatation, $\partial_x f$
	False	Print partial derivatives with standard sympy latex formatting $\frac{\partial f}{\partial x}$
ipy	*False*	Redirect print output to file for post-processing by latex
	True	Do not redirect print output (This is used for Ipython with MathJax)

`sympy.galgebra.printing.`**`xdvi`**(*filename=None, pdf='', debug=False, paper=(14, 11)*)

This function from the *galgebra/printing* module post-processes the output captured from print statements. Write the resulting latex strings to the file *filename*, processes the file with pdflatex, and displays the resulting pdf file. *pdf* is the name of the pdf viewer on your computer. If you are running *ubuntu* the *evince* viewer is automatically used. On other operating systems if *pdf* = '' the name of the pdf file is executed. If the pdf file type is associated with a viewer this will launch the viewer with the associated file. All latex files except the pdf file are deleted. If *debug = True* the file *filename* is printed to standard output for debugging purposes and *filename* (the tex file) is saved.

If *filename* is not entered the default filename is the root name of the python program being executed with *.tex* appended. The format for the *paper* is

paper=(w,h)	*w* is paper width in inches and,*h* is paper height in inches
paper='letter'	paper is standard leter size 8.5 in $\times$ 11 in

The default of *paper=(14,11)* was chosen so that long multivector expressions would not be truncated on the display.

The **xdvi** function requires that latex and a pdf viewer be installed on the computer.

As an example of using the latex printing options when the following code is executed

```
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *
Format()
(ex,ey,ez) = MV.setup('e*x|y|z')
A = MV('A','mv')
print r'\bm{A} =',A
A.Fmt(2,r'\bm{A}')
A.Fmt(3,r'\bm{A}')

xdvi()
```

The following is displayed

$$\boldsymbol{A} = A + A^{x}\boldsymbol{e_{x}} + A^{y}\boldsymbol{e_{y}} + A^{z}\boldsymbol{e_{z}} + A^{xy}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + A^{xz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} + A^{yz}\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} + A^{xyz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}}$$

$$\begin{aligned}\boldsymbol{A} = & A \\ & + A^{x}\boldsymbol{e_{x}} + A^{y}\boldsymbol{e_{y}} + A^{z}\boldsymbol{e_{z}} \\ & + A^{xy}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + A^{xz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} + A^{yz}\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \\ & + A^{xyz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}}\end{aligned}$$

$$\begin{aligned}\boldsymbol{A} = & A \\ & + A^{x}\boldsymbol{e_{x}} \\ & + A^{y}\boldsymbol{e_{y}} \\ & + A^{z}\boldsymbol{e_{z}} \\ & + A^{xy}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} \\ & + A^{xz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} \\ & + A^{yz}\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \\ & + A^{xyz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}}\end{aligned}$$

For the cases of derivatives the code is

```
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *

Format()
X = (x,y,z) = symbols('x y z')
(ex,ey,ez,grad) = MV.setup('e_x e_y e_z',metric='[1,1,1]',coords=X)

f = MV('f','scalar',fct=True)
A = MV('A','vector',fct=True)
B = MV('B','grade2',fct=True)

print r'\bm{A} =',A
```

```
print r'\bm{B} =',B

print 'grad*f =',grad*f
print r'grad|\bm{A} =',grad|A
print r'grad*\bm{A} =',grad*A

print r'-I*(grad^\bm{A}) =',-MV.I*(grad^A)
print r'grad*\bm{B} =',grad*B
print r'grad^\bm{B} =',grad^B
print r'grad|\bm{B} =',grad|B

xdvi()
```

and the latex displayed output is (f is a scalar function)

$$
\begin{aligned}
\boldsymbol{A} =& A^{x}\boldsymbol{e_{x}} + A^{y}\boldsymbol{e_{y}} + A^{z}\boldsymbol{e_{z}} \\
\boldsymbol{B} =& B^{xy}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + B^{xz}\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} + B^{yz}\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \\
\boldsymbol{\nabla} f =& \partial_{x} f\boldsymbol{e_{x}} + \partial_{y} f\boldsymbol{e_{y}} + \partial_{z} f\boldsymbol{e_{z}} \\
\boldsymbol{\nabla}\cdot\boldsymbol{A} =& \partial_{x} A^{x} + \partial_{y} A^{y} + \partial_{z} A^{z} \\
\boldsymbol{\nabla}\boldsymbol{A} =& \partial_{x} A^{x} + \partial_{y} A^{y} + \partial_{z} A^{z} + \left(-\partial_{y} A^{x} + \partial_{x} A^{y}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}} + \left(-\partial_{z} A^{x} + \partial_{x} A^{z}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{z}} \\
&+ \left(-\partial_{z} A^{y} + \partial_{y} A^{z}\right)\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \\
-I(\boldsymbol{\nabla}\wedge\boldsymbol{A}) =& \left(-\partial_{z} A^{y} + \partial_{y} A^{z}\right)\boldsymbol{e_{x}} + \left(\partial_{z} A^{x} - \partial_{x} A^{z}\right)\boldsymbol{e_{y}} + \left(-\partial_{y} A^{x} + \partial_{x} A^{y}\right)\boldsymbol{e_{z}} \\
\boldsymbol{\nabla}\boldsymbol{B} =& \left(-\partial_{y} B^{xy} - \partial_{z} B^{xz}\right)\boldsymbol{e_{x}} + \left(\partial_{x} B^{xy} - \partial_{z} B^{yz}\right)\boldsymbol{e_{y}} + \left(\partial_{x} B^{xz} + \partial_{y} B^{yz}\right)\boldsymbol{e_{z}} \\
&+ \left(\partial_{z} B^{xy} - \partial_{y} B^{xz} + \partial_{x} B^{yz}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \\
\boldsymbol{\nabla}\wedge\boldsymbol{B} =& \left(\partial_{z} B^{xy} - \partial_{y} B^{xz} + \partial_{x} B^{yz}\right)\boldsymbol{e_{x}}\wedge\boldsymbol{e_{y}}\wedge\boldsymbol{e_{z}} \\
\boldsymbol{\nabla}\cdot\boldsymbol{B} =& \left(-\partial_{y} B^{xy} - \partial_{z} B^{xz}\right)\boldsymbol{e_{x}} + \left(\partial_{x} B^{xy} - \partial_{z} B^{yz}\right)\boldsymbol{e_{y}} + \left(\partial_{x} B^{xz} + \partial_{y} B^{yz}\right)\boldsymbol{e_{z}}
\end{aligned}
$$

This example also demonstrates several other features of the latex printer. In the case that strings are input into the latex printer such as `r'grad*\boldsymbol{A}'`, `r'grad^\boldsymbol{A}'`, or `r'grad*\boldsymbol{A}'`. The text symbols *grad*, ^, |, and * are mapped by the *xdvi()* post-processor as follows if the string contains an =.

original	replacement	displayed latex
grad*A	\boldsymbol{\nabla}A	$\boldsymbol{\nabla}A$
A^B	A\wedge B	$A\wedge B$
A|B	A\cdot B	$A\cdot B$
A*B	AB	AB
A<B	A\lfloor B	$A\lfloor B$
A>B	A\rfloor B	$A\rfloor B$

If the string to be printed contains a % none of the above substitutions are made before the latex processor is applied. In general for the latex printer strings are assumed to be in a math environment (*equation** or *align**) unless the string contains a #.

Note: Except where noted the conventions for latex printing follow those of the latex printing module of sympy. This includes translating sympy variables with Greek name (such as `alpha`) to the equivalent Greek symbol (α) for the purpose of latex printing. Also a single underscore in the variable name (such as `X_j`) indicates a subscript (X_j), and a double underscore (such as `X__k`) a superscript (X^k). The only other change with regard to the sympy latex printer is that matrices are printed full size (equation displaystyle).

Printer Redirection

In order to print transparently, that is to simply use the *print* statement with both text and LaTeX printing the printer output is redirected. In the case of text printing the reason for redirecting the printer output is because the *sympy* printing functions *_print_Derivative* and *_print_Function* are redefined to make the output more compact. If one does not wish to use the compact notation redirection is not required for the text printer. If one wishes to use the redefined *_print_Derivative* and *_print_Function* the printer should be redirected with the function *ga_print_on()* and restored with the function *ga_print_off()*. Both functions can be imported from *sympy.galgebra.ga* (see *examples/galgebra/terminal_check.py* for usage).

SymPy provides a context handler that will allow you to rewrite any *ga_print_on(); "do something"; ga_print_off();* sequence like this:

```
with GA_printer:
  "do something"
```

This has the advantage that even in the case of an exception inside "do something", the *ga_print_off();* call will be made.

For LaTeX printing the *Format()* (import from *sympy.galgebra.ga*) redirects the printer output to a string. After all printing requests one must call the function *xdvi()* (import from *sympy.galgebra.printing*) tp process the string to a LaTeX format, compile with pdflatex, and displayed the resulting pdf file. The function *xdvi()* also restores the printer output to normal for standard *sympy* printing. If *Format(ipy=True)* is used there is no printer redirection and the LaTeX output is simply sent to *sys.stdout* for use in *Ipython* (*Ipython* LaTeX interface for *galgebra* not yet implemented).

Other Printing Functions

These functions are used together if one wishes to print both code and output in a single file. They work for text printing and for latex printing.

For these functions to work properly the last function defined must not contain a *Print_Function()* call (the last function defined is usually a *dummy()* function that does nothing).

Additionally, to work properly none of the functions containing *Print_Function()* can contain function definintions (local functions).

`sympy.galgebra.printing.`**`Get_Program`**`(`*off=False*`)`

Tells program to print both code and output from functions that have been properly tagged with *Print_Function()*. *Get_Program()* must be in main program before the functions that you wish code printing from are executed. the *off* argument in *Get_Program()* allows one to turn off the printing of the code by changing one line in the entire program (*off=True*).

`sympy.galgebra.printing.`**`Print_Function`**`()`

Print_Function() is included in those functions where one wishes to print the code block along with (before) the usual printed output. The *Print_Function()* statement should be included immediately after the function def statement. For proper usage of both *Print_Function()* and *Get_Program()* see the following example.

As an example consider the following code

```
from sympy.galgebra.printing import xdvi,Get_Program,Print_Function
from sympy.galgebra.ga import *
```

```
Format()

def basic_multivector_operations_3D():
    Print_Function()
    (ex,ey,ez) = MV.setup('e*x|y|z')

    A = MV('A','mv')

    A.Fmt(1,'A')
    A.Fmt(2,'A')
    A.Fmt(3,'A')

    A.even().Fmt(1,'%A_{+}')
    A.odd().Fmt(1,'%A_{-}')

    X = MV('X','vector')
    Y = MV('Y','vector')

    print 'g_{ij} =',MV.metric

    X.Fmt(1,'X')
    Y.Fmt(1,'Y')

    (X*Y).Fmt(2,'X*Y')
    (X^Y).Fmt(2,'X^Y')
    (X|Y).Fmt(2,'X|Y')
    return

def basic_multivector_operations_2D():
    Print_Function()
    (ex,ey) = MV.setup('e*x|y')

    print 'g_{ij} =',MV.metric

    X = MV('X','vector')
    A = MV('A','spinor')

    X.Fmt(1,'X')
    A.Fmt(1,'A')

    (X|A).Fmt(2,'X|A')
    (X<A).Fmt(2,'X<A')
    (A>X).Fmt(2,'A>X')
    return

def dummy():
    return

Get_Program()

basic_multivector_operations_3D()
basic_multivector_operations_2D()

xdvi()
```

The latex output of the code is

```
def basic_multivector_operations_3D():
    Print_Function()
    (ex,ey,ez) = MV.setup('e*x|y|z')
    print 'g_{ij} =',MV.metric
    A = MV('A','mv')
    A.Fmt(1,'A')
    A.Fmt(2,'A')
    A.Fmt(3,'A')
    A.even().Fmt(1,'%A_{+}')
    A.odd().Fmt(1,'%A_{-}')
    X = MV('X','vector')
    Y = MV('Y','vector')
    X.Fmt(1,'X')
    Y.Fmt(1,'Y')
    (X*Y).Fmt(2,'X*Y')
    (X^Y).Fmt(2,'X^Y')
    (X|Y).Fmt(2,'X|Y')
    return
```

Code Output:

$$g_{ij} = \left[\begin{array}{ccc} (e_x\cdot e_x) & (e_x\cdot e_y) & (e_x\cdot e_z) \\ (e_x\cdot e_y) & (e_y\cdot e_y) & (e_y\cdot e_z) \\ (e_x\cdot e_z) & (e_y\cdot e_z) & (e_z\cdot e_z) \end{array}\right]$$

$$A = A + A^{x}\boldsymbol{e_x} + A^{y}\boldsymbol{e_y} + A^{z}\boldsymbol{e_z} + A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} + A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} + A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z} + A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}$$

$$\begin{aligned} A = & A \\ & + A^{x}\boldsymbol{e_x} + A^{y}\boldsymbol{e_y} + A^{z}\boldsymbol{e_z} \\ & + A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} + A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} + A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z} \\ & + A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z} \end{aligned}$$

$$\begin{aligned} A = & A \\ & + A^{x}\boldsymbol{e_x} \\ & + A^{y}\boldsymbol{e_y} \\ & + A^{z}\boldsymbol{e_z} \\ & + A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} \\ & + A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} \\ & + A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z} \\ & + A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z} \end{aligned}$$

$$A_{+} = A + A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y} + A^{xz}\boldsymbol{e_x}\wedge\boldsymbol{e_z} + A^{yz}\boldsymbol{e_y}\wedge\boldsymbol{e_z}$$

$$A_{-} = A^{x}\boldsymbol{e_x} + A^{y}\boldsymbol{e_y} + A^{z}\boldsymbol{e_z} + A^{xyz}\boldsymbol{e_x}\wedge\boldsymbol{e_y}\wedge\boldsymbol{e_z}$$

$$X = X^{x}\boldsymbol{e_x} + X^{y}\boldsymbol{e_y} + X^{z}\boldsymbol{e_z}$$

$$Y = Y^{x}\boldsymbol{e_x} + Y^{y}\boldsymbol{e_y} + Y^{z}\boldsymbol{e_z}$$

$$\begin{aligned} XY = & X^{x}Y^{x}\left(e_x\cdot e_x\right) + X^{x}Y^{y}\left(e_x\cdot e_y\right) + X^{x}Y^{z}\left(e_x\cdot e_z\right) + X^{y}Y^{x}\left(e_x\cdot e_y\right) + X^{y}Y^{y}\left(e_y\cdot e_y\right) + X^{y}Y^{z}\left(e_y\cdot e_z\right) + X^{z}Y^{x}\left(e_x\cdot e_z\right) + X^{z}Y^{y}\left(e_y\cdot e_z\right) + X^{z}Y^{z}\left(e_z\cdot e_z\right) \\ & + \left(X^{x}Y^{y} - X^{y}Y^{x}\right)\boldsymbol{e_x}\wedge\boldsymbol{e_y} + \left(X^{x}Y^{z} - X^{z}Y^{x}\right)\boldsymbol{e_x}\wedge\boldsymbol{e_z} + \left(X^{y}Y^{z} - X^{z}Y^{y}\right)\boldsymbol{e_y}\wedge\boldsymbol{e_z} \end{aligned}$$

$$X\wedge Y = \left(X^{x}Y^{y} - X^{y}Y^{x}\right)\boldsymbol{e_x}\wedge\boldsymbol{e_y} + \left(X^{x}Y^{z} - X^{z}Y^{x}\right)\boldsymbol{e_x}\wedge\boldsymbol{e_z} + \left(X^{y}Y^{z} - X^{z}Y^{y}\right)\boldsymbol{e_y}\wedge\boldsymbol{e_z}$$

$$X\cdot Y = X^{x}Y^{x}\left(e_x\cdot e_x\right) + X^{x}Y^{y}\left(e_x\cdot e_y\right) + X^{x}Y^{z}\left(e_x\cdot e_z\right) + X^{y}Y^{x}\left(e_x\cdot e_y\right) + X^{y}Y^{y}\left(e_y\cdot e_y\right) + X^{y}Y^{z}\left(e_y\cdot e_z\right) + X^{z}Y^{x}\left(e_x\cdot e_z\right) + X^{z}Y^{y}\left(e_y\cdot e_z\right) + X^{z}Y^{z}\left(e_z\cdot e_z\right)$$

```
def basic_multivector_operations_2D():
    Print_Function()
    (ex,ey) = MV.setup('e*x|y')
    print 'g_{ij} =',MV.metric
    X = MV('X','vector')
    A = MV('A','spinor')
    X.Fmt(1,'X')
    A.Fmt(1,'A')
    (X|A).Fmt(2,'X|A')
    (X<A).Fmt(2,'X<A')
    (A>X).Fmt(2,'A>X')
    return
```

Code Output:

$$g_{ij} = \left[\begin{array}{cc} (e_x\cdot e_x) & (e_x\cdot e_y) \\ (e_x\cdot e_y) & (e_y\cdot e_y) \end{array}\right]$$

$$X = X^{x}\boldsymbol{e_x} + X^{y}\boldsymbol{e_y}$$

$$A = A + A^{xy}\boldsymbol{e_x}\wedge\boldsymbol{e_y}$$

$$X\cdot A = \left(A^{xy}\left(-X^{x}\left(e_x\cdot e_y\right) - X^{y}\left(e_y\cdot e_y\right)\right)\right)\boldsymbol{e_x} + \left(A^{xy}\left(X^{x}\left(e_x\cdot e_x\right) + X^{y}\left(e_x\cdot e_y\right)\right)\right)\boldsymbol{e_y}$$

$$X\rfloor A = \left(A^{xy}\left(-X^{x}\left(e_x\cdot e_y\right) - X^{y}\left(e_y\cdot e_y\right)\right)\right)\boldsymbol{e_x} + \left(A^{xy}\left(X^{x}\left(e_x\cdot e_x\right) + X^{y}\left(e_x\cdot e_y\right)\right)\right)\boldsymbol{e_y}$$

$$A\lfloor X = \left(A^{xy}\left(X^{x}\left(e_x\cdot e_y\right) + X^{y}\left(e_y\cdot e_y\right)\right)\right)\boldsymbol{e_x} + \left(A^{xy}\left(-X^{x}\left(e_x\cdot e_x\right) - X^{y}\left(e_x\cdot e_y\right)\right)\right)\boldsymbol{e_y}$$

Examples

Algebra

BAC-CAB Formulas This example demonstrates the most general metric tensor

$$g_{ij} = \left[\begin{array}{cccc} (a\cdot a) & (a\cdot b) & (a\cdot c) & (a\cdot d) \\ (a\cdot b) & (b\cdot b) & (b\cdot c) & (b\cdot d) \\ (a\cdot c) & (b\cdot c) & (c\cdot c) & (c\cdot d) \\ (a\cdot d) & (b\cdot d) & (c\cdot d) & (d\cdot d) \end{array}\right]$$

and how the *galgebra* module can be used to verify and expand geometric algebra identities consisting of relations between the abstract vectors a, b, c, and d.

```
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *
Format()

(a,b,c,d) = MV.setup('a b c d')
print '\\bm{a|(b*c)} =',a|(b*c)
print '\\bm{a|(b^c)} =',a|(b^c)
print '\\bm{a|(b^c^d)} =',a|(b^c^d)
print '\\bm{a|(b^c)+c|(a^b)+b|(c^a)} =',(a|(b^c))+(c|(a^b))+(b|(c^a))
print '\\bm{a*(b^c)-b*(a^c)+c*(a^b)} =',a*(b^c)-b*(a^c)+c*(a^b)
print '\\bm{a*(b^c^d)-b*(a^c^d)+c*(a^b^d)-d*(a^b^c)} =',a*(b^c^d)-b*(a^c^d)+c*(a^b^d)-d*(a^b^c)
print '\\bm{(a^b)|(c^d)} =',(a^b)|(c^d)
print '\\bm{((a^b)|c)|d} =',((a^b)|c)|d
print '\\bm{(a^b)\\times (c^d)} =',Com(a^b,c^d)

xdvi()
```

The preceeding code block also demonstrates the mapping of *, ^, and | to appropriate latex symbols.

Note: The $\times$ symbol is the commutator product of two multivectors, $A\times B = (AB - BA)/2$.

$$\begin{aligned}
\boldsymbol{a}\cdot(\boldsymbol{b}\boldsymbol{c}) &= -(a\cdot c)\,\boldsymbol{b} + (a\cdot b)\,\boldsymbol{c} \\
\boldsymbol{a}\cdot(\boldsymbol{b}\wedge\boldsymbol{c}) &= -(a\cdot c)\,\boldsymbol{b} + (a\cdot b)\,\boldsymbol{c} \\
\boldsymbol{a}\cdot(\boldsymbol{b}\wedge\boldsymbol{c}\wedge\boldsymbol{d}) &= (a\cdot d)\,\boldsymbol{b}\wedge\boldsymbol{c} - (a\cdot c)\,\boldsymbol{b}\wedge\boldsymbol{d} + (a\cdot b)\,\boldsymbol{c}\wedge\boldsymbol{d} \\
\boldsymbol{a}\cdot(\boldsymbol{b}\wedge\boldsymbol{c}) + \boldsymbol{c}\cdot(\boldsymbol{a}\wedge\boldsymbol{b}) + \boldsymbol{b}\cdot(\boldsymbol{c}\wedge\boldsymbol{a}) &= 0 \\
\boldsymbol{a}(\boldsymbol{b}\wedge\boldsymbol{c}) - \boldsymbol{b}(\boldsymbol{a}\wedge\boldsymbol{c}) + \boldsymbol{c}(\boldsymbol{a}\wedge\boldsymbol{b}) &= 3\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{c} \\
\boldsymbol{a}(\boldsymbol{b}\wedge\boldsymbol{c}\wedge\boldsymbol{d}) - \boldsymbol{b}(\boldsymbol{a}\wedge\boldsymbol{c}\wedge\boldsymbol{d}) + \boldsymbol{c}(\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{d}) - \boldsymbol{d}(\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{c}) &= 4\boldsymbol{a}\wedge\boldsymbol{b}\wedge\boldsymbol{c}\wedge\boldsymbol{d} \\
(\boldsymbol{a}\wedge\boldsymbol{b})\cdot(\boldsymbol{c}\wedge\boldsymbol{d}) &= -(a\cdot c)\,(b\cdot d) + (a\cdot d)\,(b\cdot c) \\
((\boldsymbol{a}\wedge\boldsymbol{b})\cdot\boldsymbol{c})\cdot\boldsymbol{d} &= -(a\cdot c)\,(b\cdot d) + (a\cdot d)\,(b\cdot c) \\
(\boldsymbol{a}\wedge\boldsymbol{b})\times(\boldsymbol{c}\wedge\boldsymbol{d}) &= -(b\cdot d)\,\boldsymbol{a}\wedge\boldsymbol{c} + (b\cdot c)\,\boldsymbol{a}\wedge\boldsymbol{d} + (a\cdot d)\,\boldsymbol{b}\wedge\boldsymbol{c} - (a\cdot c)\,\boldsymbol{b}\wedge\boldsymbol{d}
\end{aligned}$$

Reciprocal Frame The reciprocal frame of vectors with respect to the basis vectors is required for the evaluation of the geometric dervative. The following example demonstrates that for the case of an arbitrary 3-dimensional Euclidian basis the reciprocal basis vectors are correctly calculated.

```
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *
Format()

metric = '1 # #,'+ \
```

```
        '# 1 #,'+ \
        '# # 1,'

(e1,e2,e3) = MV.setup('e1 e2 e3',metric)

E = e1^e2^e3
Esq = (E*E).scalar()
print 'E =',E
print '%E^{2} =',Esq
Esq_inv = 1/Esq

E1 = (e2^e3)*E
E2 = (-1)*(e1^e3)*E
E3 = (e1^e2)*E

print 'E1 = (e2^e3)*E =',E1
print 'E2 =-(e1^e3)*E =',E2
print 'E3 = (e1^e2)*E =',E3

print 'E1|e2 =',(E1|e2).expand()
print 'E1|e3 =',(E1|e3).expand()
print 'E2|e1 =',(E2|e1).expand()
print 'E2|e3 =',(E2|e3).expand()
print 'E3|e1 =',(E3|e1).expand()
print 'E3|e2 =',(E3|e2).expand()
w = ((E1|e1).expand()).scalar()
Esq = expand(Esq)
print '%(E1\\cdot e1)/E^{2} =',simplify(w/Esq)
w = ((E2|e2).expand()).scalar()
print '%(E2\\cdot e2)/E^{2} =',simplify(w/Esq)
w = ((E3|e3).expand()).scalar()
print '%(E3\\cdot e3)/E^{2} =',simplify(w/Esq)

xdvi()
```

The preceeding code also demonstrated the use of the % directive in printing a string so that ^ is treated literally and not translated to *\wedge*. Note that `'%E^{2} ='` is printed as $E^2 =$

and not as $E \wedge 2 =$.

$$
\begin{aligned}
E =& \boldsymbol{e_1} \wedge \boldsymbol{e_2} \wedge \boldsymbol{e_3} \\
E^2 =& \left(e_1 \cdot e_2\right)^2 - 2\left(e_1 \cdot e_2\right)\left(e_1 \cdot e_3\right)\left(e_2 \cdot e_3\right) + \left(e_1 \cdot e_3\right)^2 + \left(e_2 \cdot e_3\right)^2 - 1 \\
E1 =& (e2 \wedge e3)E = \left(\left(e_2 \cdot e_3\right)^2 - 1\right)\boldsymbol{e_1} + \left(\left(e_1 \cdot e_2\right) - \left(e_1 \cdot e_3\right)\left(e_2 \cdot e_3\right)\right)\boldsymbol{e_2} + \left(-\left(e_1 \cdot e_2\right)\left(e_2 \cdot e_3\right) + \left(e_1 \cdot e_3\right)\right)\boldsymbol{e_3} \\
E2 =& -(e1 \wedge e3)E = \left(\left(e_1 \cdot e_2\right) - \left(e_1 \cdot e_3\right)\left(e_2 \cdot e_3\right)\right)\boldsymbol{e_1} + \left(\left(e_1 \cdot e_3\right)^2 - 1\right)\boldsymbol{e_2} + \left(-\left(e_1 \cdot e_2\right)\left(e_1 \cdot e_3\right) + \left(e_2 \cdot e_3\right)\right)\boldsymbol{e_3} \\
E3 =& (e1 \wedge e2)E = \left(-\left(e_1 \cdot e_2\right)\left(e_2 \cdot e_3\right) + \left(e_1 \cdot e_3\right)\right)\boldsymbol{e_1} + \left(-\left(e_1 \cdot e_2\right)\left(e_1 \cdot e_3\right) + \left(e_2 \cdot e_3\right)\right)\boldsymbol{e_2} + \left(\left(e_1 \cdot e_2\right)^2 - 1\right)\boldsymbol{e_3}
\end{aligned}
$$

$$
\begin{aligned}
E1 \cdot e2 =& 0 \\
E1 \cdot e3 =& 0 \\
E2 \cdot e1 =& 0 \\
E2 \cdot e3 =& 0 \\
E3 \cdot e1 =& 0 \\
E3 \cdot e2 =& 0 \\
(E1 \cdot e1)/E^2 =& 1 \\
(E2 \cdot e2)/E^2 =& 1 \\
(E3 \cdot e3)/E^2 =& 1
\end{aligned}
$$

The formulas derived for $E1$, $E2$, $E3$, and E^2 could also be applied to the numerical calculations of crystal properties.

Lorentz-Transformation A simple physics demonstation of geometric algebra is the derivation of the Lorentz-Transformation. In this demonstration a 2-dimensional Minkowski space is defined and the Lorentz-Transformation is generated from a rotation of a vector in the Minkowski space using the rotor R.

```
from sympy import symbols,sinh,cosh
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *

Format()
(alpha,beta,gamma) = symbols('alpha beta gamma')
(x,t,xp,tp) = symbols("x t x' t'")
(g0,g1) = MV.setup('gamma*t|x',metric='[1,-1]')

R = cosh(alpha/2)+sinh(alpha/2)*(g0^g1)
X = t*g0+x*g1
Xp = tp*g0+xp*g1

print 'R =',R
print r"#%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} = t'\bm{\gamma'_{t}}+x'\bm{\gamma'_{x}} "+
      r"= R\left ( t'\bm{\gamma_{t}}+x'\bm{\gamma_{x}}\right ) R^{\dagger}"

Xpp = R*Xp*R.rev()
Xpp = Xpp.collect([xp,tp])
Xpp = Xpp.subs({2*sinh(alpha/2)*cosh(alpha/2):sinh(alpha),\
               sinh(alpha/2)**2+cosh(alpha/2)**2:cosh(alpha)})
print r"%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} =",Xpp
Xpp = Xpp.subs({sinh(alpha):gamma*beta,cosh(alpha):gamma})

print r'%\f{\sinh}{\alpha} = \gamma\beta'
```

```
print r'%\f{\cosh}{\alpha} = \gamma'

print r"%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} =",Xpp.collect(gamma)

xdvi()
```

The preceeding code also demonstrates how to use the sympy *subs* functions to perform the hyperbolic half angle transformation. The code also shows the use of both the # and % directives in the text string `r"#%t\bm{\gamma_{t}}+x\bm{\gamma_{x}} = t'\bm{\gamma'_{t}}+x'\bm{\gamma'_{x}} = R\left ( t'\bm{\gamma_{t}}+x'\bm{\gamma_{x}}\right ) R^{\dagger}"`. Both the # and % are needed in this text string for two reasons. First, the text string contains an = sign. The latex preprocessor uses this a key to combine the text string with a sympy expression to be printed after the text string. The # is required to inform the preprocessor that there is no sympy expression to follow. Second, the % is requires to inform the preprocessor that the text string is to be displayed in latex math mode and not in text mode (if # is present the default latex mode is text mode unless overridden by the % directive).

$$
\begin{aligned}
R =& \cosh\left(\frac{1}{2}\alpha\right) + \sinh\left(\frac{1}{2}\alpha\right)\boldsymbol{\gamma_t} \wedge \boldsymbol{\gamma_x} \\
t\boldsymbol{\gamma_t} + x\boldsymbol{\gamma_x} =& t'\boldsymbol{\gamma'_t} + x'\boldsymbol{\gamma'_x} = R\left(t'\boldsymbol{\gamma_t} + x'\boldsymbol{\gamma_x}\right)R^{\dagger} \\
t\boldsymbol{\gamma_t} + x\boldsymbol{\gamma_x} =& \left(t'\cosh(\alpha) - x'\sinh(\alpha)\right)\boldsymbol{\gamma_t} + \left(-t'\sinh(\alpha) + x'\cosh(\alpha)\right)\boldsymbol{\gamma_x} \\
\sinh(\alpha) =& \gamma\beta \\
\cosh(\alpha) =& \gamma \\
t\boldsymbol{\gamma_t} + x\boldsymbol{\gamma_x} =& \left(\gamma\left(-\beta x' + t'\right)\right)\boldsymbol{\gamma_t} + \left(\gamma\left(-\beta t' + x'\right)\right)\boldsymbol{\gamma_x}
\end{aligned}
$$

Calculus

Derivatives in Spherical Coordinates The following code shows how to use *galgebra* to use spherical coordinates. The gradient of a scalar function, f, the divergence and curl of a vector function, A, and the exterior derivative (curl) of a bivector function, B are calculated. Note that to get the standard curl of a 3-dimension function the result is multiplied by $-I$ the negative of the pseudoscalar.

Note: In geometric calculus the operator ∇^2 is well defined on its own as the geometic derivative of the geometric derivative. However, if needed we have for the vector function A the relations (since $\nabla \cdot A$ is a scalar it's curl is equal to it's geometric derivative and it's divergence is zero) -

$$
\begin{aligned}
\nabla A =& \nabla \wedge A + \nabla \cdot A \\
\nabla^2 A =& \nabla\left(\nabla \wedge A\right) + \nabla\left(\nabla \cdot A\right) \\
\nabla^2 A =& \nabla \wedge \left(\nabla \wedge A\right) + \nabla \cdot \left(\nabla \wedge A\right) + \nabla \wedge \left(\nabla \cdot A\right) + \nabla \cdot \left(\nabla \cdot A\right) \\
\nabla^2 A =& \nabla \wedge \left(\nabla \wedge A\right) + \left(\nabla \cdot \nabla\right) A - \nabla\left(\nabla \cdot A\right) + \nabla\left(\nabla \cdot A\right) \\
\nabla^2 A =& \nabla \wedge \nabla \wedge A + \left(\nabla \cdot \nabla\right) A
\end{aligned}
$$

In the derivation we have used that $\nabla \cdot \left(\nabla \wedge A\right) = \left(\nabla \cdot \nabla\right) A - \nabla\left(\nabla \cdot A\right)$ which is implicit in the second *BAC-CAB* formula. No parenthesis is needed for the geometric curl of the curl (exterior derivative of exterior derivative) since the $\wedge$ operation is associative unlike the vector curl operator and $\nabla \cdot \nabla$ is the usual Laplacian operator.

```
from sympy import sin,cos
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *
Format()

X = (r,th,phi) = symbols('r theta phi')
curv = [[r*cos(phi)*sin(th),r*sin(phi)*sin(th),r*cos(th)],[1,r,r*sin(th)]]
(er,eth,ephi,grad) = MV.setup('e_r e_theta e_phi',metric='[1,1,1]',coords=X,curv=curv)

f = MV('f','scalar',fct=True)
A = MV('A','vector',fct=True)
B = MV('B','grade2',fct=True)

print 'A =',A
print 'B =',B

print 'grad*f =',grad*f
print 'grad|A =',grad|A
print '-I*(grad^A) =',-MV.I*(grad^A)
print 'grad^B =',grad^B

xdvi()
```

Results of code

$$
\begin{aligned}
A =& A^{r}\boldsymbol{e_{r}} + A^{\theta}\boldsymbol{e_{\theta}} + A^{\phi}\boldsymbol{e_{\phi}} \\
B =& B^{r\theta}\boldsymbol{e_{r}}\wedge\boldsymbol{e_{\theta}} + B^{r\phi}\boldsymbol{e_{r}}\wedge\boldsymbol{e_{\phi}} + B^{\theta\phi}\boldsymbol{e_{\theta}}\wedge\boldsymbol{e_{\phi}} \\
\boldsymbol{\nabla} f =& \partial_{r} f\boldsymbol{e_{r}} + \frac{\partial_{\theta} f}{r}\boldsymbol{e_{\theta}} + \frac{\partial_{\phi} f}{r\sin{(\theta)}}\boldsymbol{e_{\phi}} \\
\boldsymbol{\nabla}\cdot A =& \partial_{r} A^{r} + \frac{A^{\theta}}{r\tan{(\theta)}} + 2\frac{A^{r}}{r} + \frac{\partial_{\theta} A^{\theta}}{r} + \frac{\partial_{\phi} A^{\phi}}{r\sin{(\theta)}} \\
-I(\boldsymbol{\nabla}\wedge A) =& \left(\frac{A^{\phi}\cos{(\theta)} + \sin{(\theta)}\partial_{\theta} A^{\phi} - \partial_{\phi} A^{\theta}}{r\sin{(\theta)}}\right)\boldsymbol{e_{r}} + \left(-\partial_{r} A^{\phi} - \frac{A^{\phi}}{r} + \frac{\partial_{\phi} A^{r}}{r\sin{(\theta)}}\right)\boldsymbol{e_{\theta}} + \left(\frac{r\partial_{r} A^{\theta} + A^{\theta} - \partial_{\theta} A^{r}}{r}\right)\boldsymbol{e_{\phi}} \\
\boldsymbol{\nabla}\wedge B =& \left(\partial_{r} B^{\theta\phi} + 2\frac{B^{\theta\phi}}{r} - \frac{B^{r\phi}}{r\tan{(\theta)}} - \frac{\partial_{\theta} B^{r\phi}}{r} + \frac{\partial_{\phi} B^{r\theta}}{r\sin{(\theta)}}\right)\boldsymbol{e_{r}}\wedge\boldsymbol{e_{\theta}}\wedge\boldsymbol{e_{\phi}}
\end{aligned}
$$

Maxwell's Equations The geometric algebra formulation of Maxwell's equations is deomonstrated with the formalism developed in "Geometric Algebra for Physicists" [Doran] (page 1624). In this formalism the signature of the metric is $(1,-1,-1,-1)$ and the basis vectors are γ_t, γ_x, γ_y, and γ_z. The if $\boldsymbol{E}$ and $\boldsymbol{B}$ are the normal electric and magnetic field vectors the electric and magnetic bivectors are given by $E = \boldsymbol{E}\gamma_t$ and $B = \boldsymbol{B}\gamma_t$. The electromagnetic bivector is then $F = E + IB$ where $I = \gamma_t\gamma_x\gamma_y\gamma_z$ is the pesudo-scalar for the Minkowski space. Note that the electromagnetic bivector is isomorphic to the electromagnetic tensor. Then if J is the 4-current all of Maxwell's equations are given by $\boldsymbol{\nabla}F = J$. For more details see [Doran] (page 1624) chapter 7.

```
from sympy import symbols,sin,cos
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *

Format()

vars = symbols('t x y z')
(g0,g1,g2,g3,grad) = MV.setup('gamma*t|x|y|z',metric='[1,-1,-1,-1]',coords=vars)
```

```
I = MV.I

B = MV('B','vector',fct=True)
E = MV('E','vector',fct=True)
B.set_coef(1,0,0)
E.set_coef(1,0,0)
B *= g0
E *= g0
J = MV('J','vector',fct=True)
F = E+I*B

print 'B = \\bm{B\\gamma_{t}} =',B
print 'E = \\bm{E\\gamma_{t}} =',E
print 'F = E+IB =',F
print 'J =',J
gradF = grad*F
gradF.Fmt(3,'grad*F')

print 'grad*F = J'
(gradF.grade(1)-J).Fmt(3,'%\\grade{\\nabla F}_{1} -J = 0')
(gradF.grade(3)).Fmt(3,'%\\grade{\\nabla F}_{3} = 0')

xdvi()
```

$$
\begin{aligned}
B =& \boldsymbol{B}\boldsymbol{\gamma_t} = -B^{x}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x} - B^{y}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y} - B^{z}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_z} \\
E =& \boldsymbol{E}\boldsymbol{\gamma_t} = -E^{x}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x} - E^{y}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y} - E^{z}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_z} \\
F =& E + IB = -E^{x}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x} - E^{y}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y} - E^{z}\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_z} - B^{z}\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y} + B^{y}\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_z} - B^{x}\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
J =& J^{t}\boldsymbol{\gamma_t} + J^{x}\boldsymbol{\gamma_x} + J^{y}\boldsymbol{\gamma_y} + J^{z}\boldsymbol{\gamma_z} \\
\boldsymbol{\nabla} F =& \left(\partial_{x}E^{x} + \partial_{y}E^{y} + \partial_{z}E^{z}\right)\boldsymbol{\gamma_t} \\
&+ \left(-\partial_{z}B^{y} + \partial_{y}B^{z} - \partial_{t}E^{x}\right)\boldsymbol{\gamma_x} \\
&+ \left(\partial_{z}B^{x} - \partial_{x}B^{z} - \partial_{t}E^{y}\right)\boldsymbol{\gamma_y} \\
&+ \left(-\partial_{y}B^{x} + \partial_{x}B^{y} - \partial_{t}E^{z}\right)\boldsymbol{\gamma_z} \\
&+ \left(-\partial_{t}B^{z} + \partial_{y}E^{x} - \partial_{x}E^{y}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y} \\
&+ \left(\partial_{t}B^{y} + \partial_{z}E^{x} - \partial_{x}E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_z} \\
&+ \left(-\partial_{t}B^{x} + \partial_{z}E^{y} - \partial_{y}E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
&+ \left(\partial_{x}B^{x} + \partial_{y}B^{y} + \partial_{z}B^{z}\right)\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
\boldsymbol{\nabla} F =& J
\end{aligned}
$$

$$
\begin{aligned}
\left\langle \nabla F\right\rangle_{1} - J = 0 =& \left(-J^{t} + \partial_{x}E^{x} + \partial_{y}E^{y} + \partial_{z}E^{z}\right)\boldsymbol{\gamma_t} \\
&+ \left(-J^{x} - \partial_{z}B^{y} + \partial_{y}B^{z} - \partial_{t}E^{x}\right)\boldsymbol{\gamma_x} \\
&+ \left(-J^{y} + \partial_{z}B^{x} - \partial_{x}B^{z} - \partial_{t}E^{y}\right)\boldsymbol{\gamma_y} \\
&+ \left(-J^{z} - \partial_{y}B^{x} + \partial_{x}B^{y} - \partial_{t}E^{z}\right)\boldsymbol{\gamma_z} \\
\left\langle \nabla F\right\rangle_{3} = 0 =& \left(-\partial_{t}B^{z} + \partial_{y}E^{x} - \partial_{x}E^{y}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y} \\
&+ \left(\partial_{t}B^{y} + \partial_{z}E^{x} - \partial_{x}E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_z} \\
&+ \left(-\partial_{t}B^{x} + \partial_{z}E^{y} - \partial_{y}E^{z}\right)\boldsymbol{\gamma_t}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z} \\
&+ \left(\partial_{x}B^{x} + \partial_{y}B^{y} + \partial_{z}B^{z}\right)\boldsymbol{\gamma_x}\wedge\boldsymbol{\gamma_y}\wedge\boldsymbol{\gamma_z}
\end{aligned}
$$

Dirac Equation In [Doran] (page 1624) equation 8.89 (page 283) is the geometric algebra formulation of the Dirac equation. In this equation ψ is an 8-component real spinor which

is to say that it is a multivector with sacalar, bivector, and pseudo-vector components in the space-time geometric algebra (it consists only of even grade components).

```
from sympy import symbols,sin,cos
from sympy.galgebra.printing import xdvi
from sympy.galgebra.ga import *

Format()

vars = symbols('t x y z')
(g0,g1,g2,g3,grad) = MV.setup('gamma*t|x|y|z',metric='[1,-1,-1,-1]',coords=vars)
I = MV.I
(m,e) = symbols('m e')

psi = MV('psi','spinor',fct=True)
A = MV('A','vector',fct=True)
sig_z = g3*g0
print '\\bm{A} =',A
print '\\bm{\\psi} =',psi

dirac_eq = (grad*psi)*I*sig_z-e*A*psi-m*psi*g0
dirac_eq.simplify()

dirac_eq.Fmt(3,r'\nabla \bm{\psi} I \sigma_{z}-e\bm{A}\bm{\psi}-m\bm{\psi}\gamma_{t} = 0')

xdvi()
```

The equations displayed are the partial differential equations for each component of the Dirac equation in rectangular coordinates we the driver for the equations is the 4-potential A. One utility of these equations is to setup a numerical solver for the Dirac equation.

$$
\begin{aligned}
\boldsymbol{A} =& A^{t}\boldsymbol{\gamma_{t}} + A^{x}\boldsymbol{\gamma_{x}} + A^{y}\boldsymbol{\gamma_{y}} + A^{z}\boldsymbol{\gamma_{z}} \\
\boldsymbol{\psi} =& \psi + \psi^{tx}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}} + \psi^{ty}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{y}} + \psi^{tz}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{z}} + \psi^{xy}\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{y}} + \psi^{xz}\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{z}} + \psi^{yz}\boldsymbol{\gamma_{y}}\wedge\boldsymbol{\gamma_{z}} \\
& + \psi^{txyz}\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}}\wedge\boldsymbol{\gamma_{y}}\wedge\boldsymbol{\gamma_{z}} \\
\nabla\boldsymbol{\psi}I\sigma_{z} - e\boldsymbol{A}\boldsymbol{\psi} - m\boldsymbol{\psi}\gamma_{t} = 0 =& \left(-eA^{t}\psi - eA^{x}\psi^{tx} - eA^{y}\psi^{ty} - eA^{z}\psi^{tz} - m\psi - \partial_{y}\psi^{tx} - \partial_{z}\psi^{txyz} + \partial_{x}\psi^{ty} + \partial_{t}\psi^{xy}\right)\boldsymbol{\gamma_{t}} \\
& + \left(-eA^{t}\psi^{tx} - eA^{x}\psi - eA^{y}\psi^{xy} - eA^{z}\psi^{xz} + m\psi^{tx} + \partial_{y}\psi - \partial_{t}\psi^{ty} - \partial_{x}\psi^{xy} + \partial_{z}\psi^{yz}\right)\boldsymbol{\gamma_{x}} \\
& + \left(-eA^{t}\psi^{ty} + eA^{x}\psi^{xy} - eA^{y}\psi - eA^{z}\psi^{yz} + m\psi^{ty} - \partial_{x}\psi + \partial_{t}\psi^{tx} - \partial_{y}\psi^{xy} - \partial_{z}\psi^{xz}\right)\boldsymbol{\gamma_{y}} \\
& + \left(-eA^{t}\psi^{tz} + eA^{x}\psi^{xz} + eA^{y}\psi^{yz} - eA^{z}\psi + m\psi^{tz} + \partial_{t}\psi^{txyz} - \partial_{z}\psi^{xy} + \partial_{y}\psi^{xz} - \partial_{x}\psi^{yz}\right)\boldsymbol{\gamma_{z}} \\
& + \left(-eA^{t}\psi^{xy} + eA^{x}\psi^{ty} - eA^{y}\psi^{tx} - eA^{z}\psi^{txyz} - m\psi^{xy} - \partial_{t}\psi + \partial_{x}\psi^{tx} + \partial_{y}\psi^{ty} + \partial_{z}\psi^{tz}\right)\boldsymbol{\gamma_{t}}\wedge\boldsymbol{\gamma_{x}} \\
& + \left(-eA^{t}\psi^{xz} + eA^{x}\psi^{tz} + eA^{y}\psi^{txyz} - eA^{z}\psi^{tx} - m\psi^{xz} + \partial_{x}\psi^{txyz} + \partial_{z}\psi^{ty} - \partial_{y}\psi^{tz} - \partial_{t}\psi^{yz}\right)\boldsymbol{\gamma_{t}} \\
& + \left(-eA^{t}\psi^{yz} - eA^{x}\psi^{txyz} + eA^{y}\psi^{tz} - eA^{z}\psi^{ty} - m\psi^{yz} - \partial_{z}\psi^{tx} + \partial_{y}\psi^{txyz} + \partial_{x}\psi^{tz} + \partial_{t}\psi^{xz}\right)\boldsymbol{\gamma_{t}} \\
& + \left(-eA^{t}\psi^{txyz} - eA^{x}\psi^{yz} + eA^{y}\psi^{xz} - eA^{z}\psi^{xy} + m\psi^{txyz} + \partial_{z}\psi - \partial_{t}\psi^{tz} - \partial_{x}\psi^{xz} - \partial_{y}\psi^{yz}\right)\boldsymbol{\gamma_{x}}\wedge
\end{aligned}
$$

5.9.2 Manifold for Geometric Algebra

Class Reference

5.9.3 Vector for Geometric Algebra

Class Reference

Function Reference

5.9.4 Precedence for Geometric Algebra

Function Reference

5.9.5 Printing for Geometric Algebra

Class Reference

Function Reference

5.9.6 Noncommutative utilities for Geometric Algebra

Function Reference

5.9.7 String manipulation for Geometric Algebra

Function Reference

5.9.8 Debug code for Geometric Algebra

Function Reference

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`
- `Line`, `Ray`, `Segment`

- `Ellipse`, `Circle`
- `Polygon`, `RegularPolygon`, `Triangle`

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]
Segment(Point(0, 0), Point(1, 1/2))
>>> Segment(Point(1, S(1)/2), Point(0, 0))
Segment(Point(0, 0), Point(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
[Point(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
>>> intersection(c, l)
[Point(-5*sqrt(2)/2, -5*sqrt(2)/2), Point(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]
Segment(Point(0, 0), Point(3*a/2, b/2))

>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point(a, b/3)]
```

5.10.5 An in-depth example: Pappus' Hexagon Theorem

From Wikipedia ([WikiPappus] (page 1624)):

> 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)
>>>
>>> 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 there are no plans for extending the module to three dimensions, but it certainly would be a good addition. 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 in the plotting module entry.

5.10.8 API Reference

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.

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 467), sympy.geometry.polygon.Polygon.encloses_point (page 481)

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 438)

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 437)

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.

rotate(*angle*, *pt=None*)

Rotate `angle` radians counterclockwise about Point `pt`.

The default pt is the origin, Point(0, 0)

See Also:

`scale` (page 437), `translate` (page 437)

Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point(1, 0), Point(-1/2, sqrt(3)/2), Point(-1/2, -sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point(0, 1), Point(-sqrt(3)/2, -1/2), Point(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 437), `translate` (page 437)

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point(1, 0), Point(-1/2, sqrt(3)/2), Point(-1/2, -sqrt(3)/2))
>>> t.scale(2)
Triangle(Point(2, 0), Point(-1, sqrt(3)/2), Point(-1, -sqrt(3)/2))
>>> t.scale(2,2)
Triangle(Point(2, 0), Point(-1, sqrt(3)), Point(-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 437), `scale` (page 437)

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point(1, 0), Point(-1/2, sqrt(3)/2), Point(-1/2, -sqrt(3)/2))
>>> t.translate(2)
Triangle(Point(3, 0), Point(3/2, sqrt(3)/2), Point(3/2, -sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point(3, 2), Point(3/2, sqrt(3)/2 + 2),
    Point(3/2, -sqrt(3)/2 + 2))
```

Utils

`sympy.geometry.util.`**`intersection`**(**entities*)

The intersection of a collection of GeometryEntity instances.

Parameters **entities** : sequence of GeometryEntity

Returns **intersection** : list of GeometryEntity

Raises **NotImplementedError** :

When unable to calculate intersection.

See Also:

`sympy.geometry.entity.GeometryEntity.intersection` (page 436)

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.

Examples

```
>>> from sympy.geometry import Point, Line, Circle, intersection
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 5)
>>> l1, l2 = Line(p1, p2), Line(p3, p2)
>>> c = Circle(p2, 1)
>>> intersection(l1, p2)
[Point(1, 1)]
>>> intersection(l1, l2)
[Point(1, 1)]
>>> intersection(c, p2)
[]
>>> intersection(c, Point(1, 0))
[Point(1, 0)]
>>> intersection(c, l2)
[Point(-sqrt(5)/5 + 1, 2*sqrt(5)/5 + 1),
 Point(sqrt(5)/5 + 1, -2*sqrt(5)/5 + 1)]
```

`sympy.geometry.util.`**`convex_hull`**(**args*)

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

See Also:

sympy.geometry.point.Point (page 440), sympy.geometry.polygon.Polygon (page 477)

Notes

This can only be performed on a set of non-symbolic points.

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://softsurfer.com/Archive/algorithm_0109/algorithm_0109.htm

Examples

```
>>> from sympy.geometry import Point, convex_hull
>>> points = [(1,1), (1,2), (3,1), (-5,2), (15,4)]
>>> convex_hull(*points)
Polygon(Point(-5, 2), Point(1, 1), Point(3, 1), Point(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 436)

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 440), sympy.geometry.line.Segment (page 458), sympy.geometry.polygon.Polygon (page 477)

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
(Point(20/3, 10/3), Point(20/3, 70/3))
>>> centroid(p, q)
Point(20/3, 40/3)
>>> p, q = Segment((0, 0), (2, 0)), Segment((0, 0), (2, 2))
>>> centroid(p, q)
Point(1, -sqrt(2) + 2)
>>> centroid(Point(0, 0), Point(2, 0))
Point(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)
Point(3/2, 1/2)
>>> centroid(p, p, p, q) # centroid x-coord shifts left
Point(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)
Point(11/10, 1/2)
```

Points

class sympy.geometry.point.**Point**

A point in a 2-dimensional Euclidean space.

Parameters **coords** : sequence of 2 coordinate values.

Raises **NotImplementedError** :

When trying to create a point with more than two dimensions. When *intersection* is called with object other than a Point.

TypeError :

When trying to add or subtract points with different dimensions.

See Also:

`sympy.geometry.line.Segment` **(page 458)** Connects two Points

Notes

Currently only 2-dimensional points are supported.

Examples

```
>>> from sympy.geometry import Point
>>> from sympy.abc import x
>>> Point(1, 2)
Point(1, 2)
>>> Point([1, 2])
Point(1, 2)
>>> Point(0, x)
Point(0, x)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point(0.5, 0.25)
Point(1/2, 1/4)
>>> Point(0.5, 0.25, evaluate=False)
Point(0.5, 0.25)
```

Attributes

x	
y	
length	

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 459)

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(*p2*)

Return dot product of self with another Point.

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).

Returns **point** : Point

Examples

```
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point(1/2, 3/2)
>>> p1.evalf()
Point(0.5, 1.5)
```

intersection(*o*)

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 Point
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point(0, 0)]
```

is_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 **is_collinear** : boolean

See Also:

sympy.geometry.line.Line (page 454)

Notes

Slope is preserved everywhere on a line, so the slope between any two points on the line should be the same. Take the first two points, p1 and p2, and create a translated point v1 with p1 as the origin. Now for every other point we create a translated point, vi with p1 also as the origin. Note that these translations preserve slope since everything is consistently translated to a new origin of p1. Since slope is preserved then we have the following equality:

- v1_slope = vi_slope
- v1.y/v1.x = vi.y/vi.x (due to translation)
- v1.y*vi.x = vi.y*v1.x
- v1.y*vi.x - vi.y*v1.x = 0 (*)

Hence, if we have a vi such that the equality in (*) is False then the points are not collinear. We do this test for every point in the list, and if all pass then they are collinear.

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(**points*)

Is a sequence of points concyclic?

Test whether or not a sequence of points are concyclic (i.e., they lie on a circle).

Parameters **points** : sequence of Points

Returns **is_concyclic** : boolean

True if points are concyclic, False otherwise.

See Also:

sympy.geometry.ellipse.Circle (page 474)

Notes

No points are not considered to be concyclic. One or two points are definitely concyclic and three points are conyclic iff they are not collinear.

For more than three points, create a circle from the first three points. If the circle cannot be created (i.e., they are collinear) then all of the points cannot be concyclic. If the circle is created successfully then simply check the remaining points for containment in the circle.

Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(-1, 0), Point(1, 0)
>>> p3, p4 = Point(0, 1), Point(-1, 2)
>>> Point.is_concyclic(p1, p2, p3)
True
>>> Point.is_concyclic(p1, p2, p3, p4)
False
```

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 460)

Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(13, 5)
>>> p1.midpoint(p2)
Point(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).

Returns **point** : Point

Examples

```
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point(1/2, 3/2)
>>> p1.evalf()
Point(0.5, 1.5)
```

rotate(*angle*, *pt=None*)

Rotate `angle` radians counterclockwise about Point `pt`.

See Also:

`rotate` (page 445), `scale` (page 445)

Examples

```
>>> from sympy import Point, pi
>>> t = Point(1, 0)
>>> t.rotate(pi/2)
Point(0, 1)
>>> t.rotate(pi/2, (2, 0))
Point(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 445), `translate` (page 445)

Examples

```
>>> from sympy import Point
>>> t = Point(1, 1)
>>> t.scale(2)
Point(2, 1)
>>> t.scale(2, 2)
Point(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 445), `scale` (page 445)

Examples

```
>>> from sympy import Point
>>> t = Point(0, 1)
>>> t.translate(2)
Point(2, 1)
>>> t.translate(2, 2)
Point(2, 3)
>>> t + Point(2, 2)
Point(2, 3)
```

x

Returns the X coordinate of the Point.

Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.x
0
```

y

Returns the Y coordinate of the Point.

Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.y
1
```

Lines

class `sympy.geometry.line.`**`LinearEntity`**

An abstract base class for all linear entities (line, ray and segment) in a 2-dimensional Euclidean space.

See Also:

`sympy.geometry.entity.GeometryEntity` (page 436)

Notes

This is an abstract class and is not meant to be instantiated. Subclasses should implement the following methods:

- __eq__
- contains

Attributes

p1	
p2	
coefficients	
slope	
points	

angle_between(*l1*, *l2*)

The angle formed between the two linear entities.

Parameters **l1** : LinearEntity

l2 : LinearEntity

Returns **angle** : angle in radians

See Also:

`is_perpendicular` (page 450)

Notes

From the dot product of vectors v1 and v2 it is known that:

```
dot(v1, v2) = |v1|*|v2|*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
>>> p1, p2, p3 = Point(0, 0), Point(0, 4), Point(2, 0)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.angle_between(l2)
pi/2
```

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 440)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.arbitrary_point()
Point(4*t + 1, 3*t)
```

coefficients

The coefficients (a, b, c) for the linear equation $ax + by + c = 0$.

See Also:

sympy.geometry.line.Line.equation (page 455)

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)
```

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.

intersection(*o*)

The intersection with another geometrical entity.

Parameters **o** : Point or LinearEntity

Returns **intersection** : list of geometrical entities

See Also:

sympy.geometry.point.Point (page 440)

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)
[Point(7, 7)]

>>> p4, p5 = Point(5, 0), Point(0, 3)
>>> l2 = Line(p4, p5)
>>> l1.intersection(l2)
[Point(15/8, 15/8)]
```

```
>>> p6, p7 = Point(0, 5), Point(2, 6)
>>> s1 = Segment(p6, p7)
>>> l1.intersection(s1)
[]
```

is_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 are concurrent,

False : otherwise.

See Also:

sympy.geometry.util.intersection (page 438)

Notes

Simply take the first two lines and find their intersection. If there is no intersection, then the first two lines were parallel and had no intersection so concurrency is impossible amongst the whole set. Otherwise, check to see if the intersection point of the first two lines is a member on the rest of the lines. If so, the lines are concurrent.

Examples

```
>>> from sympy import Point, Line
>>> 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)
>>> l1.is_concurrent(l2, l3)
True

>>> l4 = Line(p2, p3)
>>> l4.is_concurrent(l2, l3)
False
```

is_parallel(*l1, l2*)

Are two linear entities parallel?

Parameters **l1** : LinearEntity

l2 : LinearEntity

Returns **True** : if l1 and l2 are parallel,

False : otherwise.

See Also:

coefficients (page 448)

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)
>>> l3 = Line(p3, p5)
>>> Line.is_parallel(l1, l3)
False
```

is_perpendicular(*l1, l2*)

Are two linear entities perpendicular?

Parameters **l1** : LinearEntity

l2 : LinearEntity

Returns **True** : if l1 and l2 are perpendicular,

False : otherwise.

See Also:

coefficients (page 448)

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
```

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 440)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p1
Point(0, 0)
```

p2

The second defining point of a linear entity.

See Also:

sympy.geometry.point.Point (page 440)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p2
Point(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 449)

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
```

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 450), perpendicular_segment (page 452)

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
```

perpendicular_segment(*p*)

Create a perpendicular line segment from p to this line.

The enpoints 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 452)

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))
Segment(Point(2, 2), Point(4, 0))
```

points

The two points used to define this linear entity.

Returns **points** : tuple of Points

See Also:

sympy.geometry.point.Point (page 440)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 11)
>>> l1 = Line(p1, p2)
>>> l1.points
(Point(0, 0), Point(5, 11))
```

projection(*o*)

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 440), perpendicular_line (page 452)

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 is done by creating a perpendicular line through P and L and finding its intersection with L.

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)
Point(1/4, 1/4)

>>> p4, p5 = Point(10, 0), Point(12, 1)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment(Point(5, 5), Point(13/2, 13/2))
```

random_point()

A random point on a LinearEntity.

Returns **point** : Point

See Also:

sympy.geometry.point.Point (page 440)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> p3 = l1.random_point()
>>> # random point - don't know its coords in advance
>>> p3
Point(...)
>>> # point should belong to the line
>>> p3 in l1
True
```

slope

The slope of this linear entity, or infinity if vertical.

Returns **slope** : number or sympy expression

See Also:

coefficients (page 448)

Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.slope
5/3

>>> p3 = Point(0, 4)
>>> l2 = Line(p1, p3)
>>> l2.slope
oo
```

class sympy.geometry.line.**Line**

An infinite line in space.

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 440)

Notes

At the moment only lines in a 2D space can be declared, because Points can be defined only for 2D spaces.

Examples

```
>>> import sympy
>>> from sympy import Point
>>> from sympy.abc import L
>>> from sympy.geometry import Line, Segment
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line(Point(2, 3), Point(3, 5))
>>> L.points
(Point(2, 3), Point(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword `slope`:

```
>>> Line(Point(0, 0), slope=0)
Line(Point(0, 0), Point(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

contains(*o*)

Return True if o is on this Line, or False otherwise.

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` (page 448)

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
```

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.point.Point (page 440), Line (page 454)

Notes

At the moment only rays in a 2D space can be declared, because Points can be defined only for 2D spaces.

Examples

```
>>> import sympy
>>> from sympy import Point, pi
>>> from sympy.abc import r
>>> from sympy.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray(Point(2, 3), Point(3, 5))
>>> r.points
(Point(2, 3), Point(3, 5))
>>> r.source
Point(2, 3)
>>> r.xdirection
```

```
oo
>>> r.ydirection
oo
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

Attributes

source	
xdirection	
ydirection	

contains(*o*)

Is other GeometryEntity contained in this Ray?

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 Point, 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 440)

Examples

```
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.source
Point(0, 0)
```

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 458)

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 457)

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.**Segment**

An undirected line segment in space.

Parameters **p1** : Point

p2 : Point

See Also:

sympy.geometry.point.Point (page 440), Line (page 454)

Notes

At the moment only segments in a 2D space can be declared, because Points can be defined only for 2D spaces.

Examples

```
>>> import sympy
>>> from sympy import Point
>>> from sympy.abc import s
>>> from sympy.geometry import Segment
```

```
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment(Point(1, 0), Point(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1))
>>> s
Segment(Point(1, 1), Point(4, 3))
>>> s.points
(Point(1, 1), Point(4, 3))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point(5/2, 2)
```

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
```

distance(*o*)

Finds the shortest distance between a line segment and a point.

Raises **NotImplementedError is raised if o 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)
```

length

The length of the line segment.

See Also:

sympy.geometry.point.Point.distance (page 441)

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.length
5
```

midpoint

The midpoint of the line segment.

See Also:

sympy.geometry.point.Point.midpoint (page 444)

Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.midpoint
Point(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 452)

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()
Line(Point(3, 3), Point(9, -3))

>>> s1.perpendicular_bisector(p3)
Segment(Point(3, 3), Point(5, 1))
```

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]
```

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 154), sympy.polys.polyfuncs.interpolate (page 1002)

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)
Point(4, 16)
>>> C.arbitrary_point(a)
Point(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 440)

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()
Point(2*t, t**2)
>>> C.arbitrary_point(C.parameter)
Point(2*s, s**2)
>>> C.arbitrary_point(None)
Point(2*s, s**2)
>>> C.arbitrary_point(Symbol('a'))
Point(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
set([a])
```

functions

The functions specifying the curve.

Returns **functions** : list of parameterized coordinate functions.

See Also:

parameter (page 463)

Examples

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.functions
(t, t**2)
```

limits

The limits for the curve.

Returns **limits** : tuple

Contains parameter and lower and upper limits.

See Also:

plot_interval (page 463)

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 462)

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 463)** 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 474)

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(Point(3, 1), 3, 9/5)
```

Plotting:

```
>>> from sympy.plotting.pygletplot import PygletPlot as Plot
>>> from sympy import Circle, Segment
>>> c1 = Circle(Point(0,0), 1)
>>> Plot(c1)
[0]: cos(t), sin(t), 'mode=parametric'
>>> p = Plot()
>>> p[0] = c1
>>> radius = Segment(c1.center, c1.random_point())
>>> p[1] = radius
>>> p
[0]: cos(t), sin(t), 'mode=parametric'
[1]: t*cos(1.5460862150362053579755183 82),
t*sin(1.5460862150362053579755183 82), 'mode=parametric'
```

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 472)** 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 440)

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.arbitrary_point()
Point(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
```

center

The center of the ellipse.

Returns **center** : number

See Also:

sympy.geometry.point.Point (page 440)

Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.center
Point(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*Integral(sqrt((-8*_x**2/9 + 1)/(-_x**2 + 1)), (_x, 0, 1))
```

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 440)

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 466) 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
```

foci

The foci of the ellipse.

Raises **ValueError** :

When the major and minor axis cannot be determined.

See Also:

sympy.geometry.point.Point (page 440)

focus_distance (page 469) 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
(Point(-2*sqrt(2), 0), Point(2*sqrt(2), 0))
```

focus_distance

The focale distance of the ellipse.

The distance between the center and one focus.

Returns **focus_distance** : number

See Also:

foci (page 468)

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 474), major (page 471), minor (page 471)

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 436)

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))
[Point(5, 0)]
>>> e.intersection(Line(Point(0,0), Point(0, 1)))
[Point(0, -7), Point(0, 7)]
>>> e.intersection(Line(Point(5,0), Point(5, 1)))
[Point(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))
[Point(0, -3*sqrt(15)/4), Point(0, 3*sqrt(15)/4)]
>>> e.intersection(Ellipse(Point(5, 0), 4, 3))
[Point(2, -3*sqrt(7)/4), Point(2, 3*sqrt(7)/4)]
>>> e.intersection(Ellipse(Point(100500, 0), 4, 3))
[]
>>> e.intersection(Ellipse(Point(0, 0), 3, 4))
[Point(-363/175, -48*sqrt(111)/175), Point(-363/175, 48*sqrt(111)/175),
Point(3, 0)]
>>> e.intersection(Ellipse(Point(-1, 0), 3, 4))
[Point(-17/5, -12/5), Point(-17/5, 12/5), Point(7/5, -12/5),
Point(7/5, 12/5)]
```

is_tangent(*o*)

Is *o* tangent to the ellipse?

Parameters **o** : GeometryEntity

An Ellipse, LinearEntity or Polygon

Returns **is_tangent: boolean** :

True if o is tangent to the ellipse, False otherwise.

Raises **NotImplementedError** :

When the wrong type of argument is supplied.

See Also:

tangent_lines (page 474)

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 469), vradius (page 474), minor (page 471)

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 469), vradius (page 474), major (page 471)

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
```

periapsis

The periapsis of the ellipse.

The shortest distance between the focus and the contour.

Returns **periapsis** : number

See Also:

apoapsis **(page 466)** 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 440)

arbitrary_point **(page 466)** Returns parameterized point on ellipse

Notes

An arbitrary_point with a random value of t substituted into it may not test as being on the ellipse because the expression tested that a point is on the ellipse doesn't simplify to zero and doesn't evaluate exactly to zero:

```
>>> from sympy.abc import t
>>> e1.arbitrary_point(t)
Point(3*cos(t), 2*sin(t))
>>> p2 = _.subs(t, 0.1)
>>> p2 in e1
False
```

Note that arbitrary_point routine does not take this approach. A value for cos(t) and sin(t) (not t) is substituted into the arbitrary point. There is a small chance that this will give a point that will not test as being in the ellipse, so the process is repeated (up to 10 times) until a valid point is obtained.

Examples

```
>>> from sympy import Point, Ellipse, Segment
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.random_point() # gives some random point
Point(...)
>>> p1 = e1.random_point(seed=0); p1.n(2)
Point(2.1, 1.4)
```

The random_point method assures that the point will test as being in the ellipse:

```
>>> p1 in e1
True
```

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(Point(1, 0), -1)
```

rotate(*angle=0, pt=None*)

Rotate `angle` radians counterclockwise about Point `pt`.

Note: since the general ellipse is not supported, the axes of the ellipse will not be rotated. Only the center is rotated to a new position.

Examples

```
>>> from sympy import Ellipse, pi
>>> Ellipse((1, 0), 2, 1).rotate(pi/2)
Ellipse(Point(0, 1), 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(Point(0, 0), 4)
>>> Ellipse((0, 0), 2, 1).scale(2)
Ellipse(Point(0, 0), 4, 1)
```

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 440), sympy.geometry.line.Line (page 454)

Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.tangent_lines(Point(3, 0))
[Line(Point(3, 0), Point(3, -12))]

>>> # This will plot an ellipse together with a tangent line.
>>> from sympy.plotting.pygletplot import PygletPlot as Plot
>>> from sympy import Point, Ellipse
>>> e = Ellipse(Point(0,0), 3, 2)
>>> t = e.tangent_lines(e.random_point())
>>> p = Plot()
>>> p[0] = e
>>> p[1] = t
```

vradius

The vertical radius of the ellipse.

Returns **vradius** : number

See Also:

hradius (page 469), major (page 471), minor (page 471)

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 464), sympy.geometry.point.Point (page 440)

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 costructed 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, Point(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)
[Point(5, 0)]
>>> c1.intersection(Ray(p1, p2))
[Point(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 471), `Ellipse.minor` (page 471), `Ellipse.hradius` (page 469), `Ellipse.vradius` (page 474)

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(Point(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(Point(0, 0), 2)
>>> Circle((0, 0), 1).scale(2, 4)
Ellipse(Point(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.

If the Polygon has intersecting sides.

See Also:

sympy.geometry.point.Point (page 440), sympy.geometry.line.Segment (page 458), Triangle (page 491)

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(Point(0, 0), Point(1, 0), Point(5, 1), Point(0, 1))
>>> Polygon(p1, p2)
Segment(Point(0, 0), Point(1, 0))
>>> Polygon(p1, p2, p5)
Segment(Point(0, 0), Point(3, 0))
```

While the sides of a polygon are not allowed to cross implicitly, they can do so explicitly. For example, a polygon shaped like a Z with the top left connecting to the bottom right of the Z must have the point in the middle of the Z explicitly given:

```
>>> mid = Point(1, 1)
>>> Polygon((0, 2), (2, 2), mid, (0, 0), (2, 0), mid).area
0
>>> Polygon((0, 2), (2, 2), mid, (2, 0), (0, 0), mid).area
-2
```

When the 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 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(Point(0, 0), 1, 3, 0)
>>> p.vertices[0]
Point(1, 0)
>>> p.args[0]
Point(0, 0)
>>> p.spin(pi/2)
>>> p.vertices[0]
Point(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 440), `sympy.geometry.line.LinearEntity.angle_between` (page 447)

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 440)

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)
Point(1, 1/2)
```

area

The area of the polygon.

See Also:

sympy.geometry.ellipse.Ellipse.area (page 466)

Notes

The area calculation can be positive or negative based on the orientation of the points.

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
```

centroid

The centroid of the polygon.

Returns **centroid** : Point

See Also:

sympy.geometry.point.Point (page 440), sympy.geometry.util.centroid (page 440)

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
Point(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 440), sympy.geometry.ellipse.Ellipse.encloses_point (page 467)

Notes

Being on the border of self is considered False.

References

[1] http://www.ariel.com.au/a/python-point-int-poly.html

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 two polygons.

The intersection may be empty and can contain individual Points and complete Line Segments.

Parameters **other: Polygon** :

Returns **intersection** : list

The list of Segments and Points

See Also:

sympy.geometry.point.Point (page 440), sympy.geometry.line.Segment (page 458)

Examples

```
>>> from sympy import Point, Polygon
>>> 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)
[Point(2/3, 0), Point(9/5, 1/5), Point(7/3, 1), Point(1/3, 1)]
```

is_convex()

Is the polygon convex?

A polygon is convex if all its interior angles are less than 180 degrees.

Returns **is_convex** : boolean

True if this polygon is convex, False otherwise.

See Also:

sympy.geometry.util.convex_hull (page 438)

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 459)

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]
```

sides

The line segments that form the sides of the polygon.

Returns **sides** : list of sides

Each side is a Segment.

See Also:

sympy.geometry.point.Point (page 440), sympy.geometry.line.Segment (page 458)

Notes

The Segments that represent the sides are an undirected line segment so cannot be used to tell the orientation of the polygon.

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
[Segment(Point(0, 0), Point(1, 0)),
Segment(Point(1, 0), Point(5, 1)),
Segment(Point(0, 1), Point(5, 1)), Segment(Point(0, 0), Point(0, 1))]
```

vertices

The vertices of the polygon.

Returns **vertices** : tuple of Points

See Also:

sympy.geometry.point.Point (page 440)

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
(Point(0, 0), Point(1, 0), Point(5, 1), Point(0, 1))
>>> poly.args[0]
Point(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 440), Polygon (page 477)

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(Point(0, 0), 5, 3, 0)
>>> r.vertices[0]
Point(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
{Point(-5/2, -5*sqrt(3)/2): pi/3,
 Point(-5/2, 5*sqrt(3)/2): pi/3,
 Point(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 459), `sympy.geometry.ellipse.Circle.radius` (page 476)

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
(Point(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 440), sympy.geometry.ellipse.Ellipse.center (page 467)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point(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 440), sympy.geometry.ellipse.Ellipse.center (page 467)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point(0, 0)
```

circumcenter

Alias for center.

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.circumcenter
Point(0, 0)
```

circumcircle

The circumcircle of the RegularPolygon.

Returns **circumcircle** : Circle

See Also:

circumcenter (page 486), sympy.geometry.ellipse.Circle (page 474)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.circumcircle
Circle(Point(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 467)

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 447)

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 488), sympy.geometry.ellipse.Circle (page 474)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 7)
>>> rp.incircle
Circle(Point(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 447)

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 459), sympy.geometry.ellipse.Circle.radius (page 476)

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.

```
>>> from sympy import RegularPolygon, Line

>>> RegularPolygon((0, 0), 1, 4).reflect(Line((0, 1), slope=-2))
RegularPolygon(Point(4/5, 2/5), -1, 4, acos(3/5))
```

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
Point(2, 0)
>>> t.rotate(pi/2).vertices[0] # vertex on y axis now
Point(0, 2)
```

See Also:

rotation (page 490)

spin **(page 490)** 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.geometry import RegularPolygon, Point
>>> RegularPolygon(Point(0, 0), 3, 4, pi).rotation
pi
```

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(Point(0, 0), 2, 4, 0)
```

Asymmetric scaling returns a kite as a Polygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 1)
Polygon(Point(2, 0), Point(0, 1), Point(-2, 0), Point(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]
Point(1, 0)
>>> r.spin(pi/6)
>>> r.vertices[0]
Point(sqrt(3)/2, 1/2)
```

See Also:

rotation (page 490)

rotate (page 490) 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 440)

Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.vertices
[Point(5, 0), Point(0, 5), Point(-5, 0), Point(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 440), Polygon (page 477)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
Triangle(Point(0, 0), Point(4, 0), Point(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(Point(0, 0), Point(3, 0), Point(3, 4))
>>> Triangle(asa=(30, 1, 30))
```

```
Triangle(Point(0, 0), Point(1, 0), Point(1/2, sqrt(3)/6))
>>> Triangle(sas=(1, 45, 2))
Triangle(Point(0, 0), Point(2, 0), Point(sqrt(2)/2, sqrt(2)/2))
```

Attributes

vertices	
altitudes	
orthocenter	
circumcenter	
circumradius	
circumcircle	
inradius	
incircle	
medians	
medial	

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 440), sympy.geometry.line.Segment.length (page 459)

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]
Segment(Point(0, 0), Point(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 440), sympy.geometry.line.Segment (page 458)

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(0, sqrt(2) - 1), Point(1, 0))
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 440)

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
Point(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 474)

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(Point(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 476)

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)
```

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 494), `sympy.geometry.point.Point` (page 440)

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
Point(-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 474)

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(Point(-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 494), `sympy.geometry.ellipse.Circle.radius` (page 476)

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 436), RegularPolygon (page 484), is_isosceles (page 495), is_right (page 495), is_scalene (page 496)

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 495), is_right (page 495), is_scalene (page 496)

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 450), is_equilateral (page 495), is_isosceles (page 495), is_scalene (page 496)

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 495), is_isosceles (page 495), is_right (page 495)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(1, 4))
>>> t1.is_scalene()
True
```

is_similar(*t1, 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 436)

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 460)

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(Point(1/2, 0), Point(1/2, 1/2), Point(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 444), `sympy.geometry.line.Segment.midpoint` (page 460)

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]
Segment(Point(0, 0), Point(1/2, 1/2))
```

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 440)

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
Point(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 440)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t.vertices
(Point(0, 0), Point(4, 0), Point(4, 3))
```

5.11 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 523)

- `integrate(f, x)` returns the indefinite integral $\int f\,dx$
- `integrate(f, (x, a, b))` returns the definite integral $\int_a^b f\,dx$

5.11.1 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 *e *sin(x)   x *e *cos(x)      x          e *sin(x)   e *cos(x)
------------ + ------------ - x*e *sin(x) + --------- - ---------
     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
```

5.11.2 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 $f(x)$,

$$F(s) = \int_0^\infty x^{s-1} f(x) \mathrm{d}x.$$

For all "sensible" functions, this converges absolutely in a strip $a < \mathrm{Re}(s) < b$.

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), 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 499), `laplace_transform` (page 500), `fourier_transform` (page 501), `hankel_transform` (page 504), `inverse_hankel_transform` (page 505)

`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 `strip=(a, b)`.

This can be defined as

$$f(x) = \int_{c-i\infty}^{c+i\infty} x^{-s} F(s) \mathrm{d}s,$$

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 499), `hankel_transform` (page 504), `inverse_hankel_transform` (page 505)

`sympy.integrals.transforms.`**`laplace_transform`**(*f, t, s, **hints*)

Compute the Laplace Transform $F(s)$ of $f(t)$,

$$F(s) = \int_0^\infty e^{-st} f(t) \mathrm{d}t.$$

For all "sensible" functions, this converges absolutely in a half plane $a < \mathrm{Re}(s)$.

This function returns `(F, a, cond)` where `F` is the Laplace transform of `f`, $\mathrm{Re}(s) > a$ 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 500), `mellin_transform` (page 499), `fourier_transform` (page 501), `hankel_transform` (page 504), `inverse_hankel_transform` (page 505)

sympy.integrals.transforms.**inverse_laplace_transform**(*F, s, t, plane=None, **hints*)

Compute the inverse Laplace transform of $F(s)$, defined as

$$f(t) = \int_{c-i\infty}^{c+i\infty} e^{st} F(s) \mathrm{d}s,$$

for c so large that $F(s)$ has no singularites in the half-plane $\mathrm{Re}(s) > c - \epsilon$.

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 500), `hankel_transform` (page 504), `inverse_hankel_transform` (page 505)

sympy.integrals.transforms.**fourier_transform**(*f, x, k, **hints*)

Compute the unitary, ordinary-frequency Fourier transform of f, defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} \mathrm{d}x.$$

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 502), sine_transform (page 502), inverse_sine_transform (page 503), cosine_transform (page 503), inverse_cosine_transform (page 504), hankel_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**inverse_fourier_transform**(*F*, *k*, *x*, ***hints*)

Compute the unitary, ordinary-frequency inverse Fourier transform of F, defined as

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i x k} \mathrm{d}k.$$

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 501), sine_transform (page 502), inverse_sine_transform (page 503), cosine_transform (page 503), inverse_cosine_transform (page 504), hankel_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**sine_transform**(*f*, *x*, *k*, ***hints*)

Compute the unitary, ordinary-frequency sine transform of f, defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_{0}^{\infty} f(x) \sin(2\pi x k) \mathrm{d}x.$$

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 501), inverse_fourier_transform (page 502), inverse_sine_transform (page 503), cosine_transform (page 503), inverse_cosine_transform (page 504), hankel_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**inverse_sine_transform**(*F*, *k*, *x*, ***hints*)

Compute the unitary, ordinary-frequency inverse sine transform of F, defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^\infty F(k) \sin(2\pi x k) \mathrm{d}k.$$

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 501), inverse_fourier_transform (page 502), sine_transform (page 502), cosine_transform (page 503), inverse_cosine_transform (page 504), hankel_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**cosine_transform**(*f*, *x*, *k*, ***hints*)

Compute the unitary, ordinary-frequency cosine transform of f, defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^\infty f(x) \cos(2\pi x k) \mathrm{d}x.$$

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 501), inverse_fourier_transform (page 502), sine_transform (page 502), inverse_sine_transform (page 503), inverse_cosine_transform (page 504), hankel_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**inverse_cosine_transform**(*F*, *k*, *x*, ***hints*)

Compute the unitary, ordinary-frequency inverse cosine transform of F, defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^\infty F(k) \cos(2\pi x k) \mathrm{d}k.$$

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 501), inverse_fourier_transform (page 502), sine_transform (page 502), inverse_sine_transform (page 503), cosine_transform (page 503), hankel_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**hankel_transform**(*f*, *r*, *k*, *nu*, ***hints*)

Compute the Hankel transform of f, defined as

$$F_\nu(k) = \int_0^\infty f(r) J_\nu(kr) r \mathrm{d}r.$$

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 501), inverse_fourier_transform (page 502), sine_transform (page 502), inverse_sine_transform (page 503), cosine_transform (page 503), inverse_cosine_transform (page 504), inverse_hankel_transform (page 505), mellin_transform (page 499), laplace_transform (page 500)

sympy.integrals.transforms.**inverse_hankel_transform**(*F*, *k*, *r*, *nu*, ***hints*)

Compute the inverse Hankel transform of F defined as

$$f(r) = \int_0^\infty F_\nu(k) J_\nu(kr) k \mathrm{d}k.$$

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 501), inverse_fourier_transform (page 502), sine_transform (page 502), inverse_sine_transform (page 503), cosine_transform (page 503), inverse_cosine_transform (page 504), hankel_transform (page 504), mellin_transform (page 499), laplace_transform (page 500)

5.11.3 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 do 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 $\int f(x)\mathrm{d}x$ or $\int_0^\infty f(x)\mathrm{d}x$ 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. Morever, 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, ± 1, ∞). 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 $\mathcal{S} = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : \mathcal{S} :\to \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives $\mathcal{S}$ 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 $\mathrm{Exp} : \mathbb{C} \to \mathcal{S}, (x + iy) \mapsto (\exp(x), y)$ and $\mathrm{Log} : \mathcal{S} \to \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point "zero" (0) to $\mathcal{S}$ and denote the resulting object $\mathcal{S}_0$. Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by $\{\mathrm{Exp}(z) : \Re(z) < k\}$, which at least defines a topology. Elements of

$\mathcal{S}_0$ shall be called polar numbers. We further define functions $\text{Arg} : \mathcal{S} \to \mathbb{R}, (r, \theta) \mapsto \theta$ and $|.| : \mathcal{S}_0 \to \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from $\mathbb{C}$ to $\mathcal{S}$. We define $\text{Exp}(a)\,\text{Exp}(b) = \text{Exp}(a + b)$ for $a, b \in \mathbb{C}$, also for $a \in \mathcal{S}$ 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 $f : \mathcal{S} \to \mathbb{C}$ 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 knows 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" $C = \mathbb{C} \backslash \mathbb{R}_{\leq 0}$. Define a function $l : C \to \mathcal{S}$ 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 minimise (a) the number of powers occuring in $f_i(x)$ and (b) the number of different functions occuring 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 analyitc.)

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 1624)).

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 is satisfies $G(\mathrm{Exp}(2\pi i)z) = G(z)$ 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 staded 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. $$\int_0^\infty G_{p,q}^{m,n}\left(\begin{matrix} a_1, \ldots, a_p \\ b_1, \ldots, b_q \end{matrix}\middle| \eta x\right) \mathrm{d}x = \frac{\prod_{j=1}^m \Gamma(b_j+1) \prod_{j=1}^n \Gamma(-a_j)}{\eta \prod_{j=m+1}^q \Gamma(-b_j) \prod_{j=n+1}^p \Gamma(a_j+1)}$$

2. $$\int_0^\infty G_{u,v}^{s,t}\left(\begin{matrix} c_1, \ldots, c_u \\ d_1, \ldots, d_v \end{matrix}\middle| \sigma x\right) G_{p,q}^{m,n}\left(\begin{matrix} a_1, \ldots, a_p \\ b_1, \ldots, b_q \end{matrix}\middle| \omega x\right) \mathrm{d}x = G_{v+p,u+q}^{m+t,n+s}\left(\begin{matrix} a_1, \ldots, a_n, -d_1, \ldots, -d_v, a_{n+1}, \ldots, a_p \\ b_1, \ldots, b_m, -c_1, \ldots, -c_u, b_{m+1}, \ldots, b_q \end{matrix}\middle| \frac{\omega}{\sigma}\right)$$

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 1627) and [Luke1969] (page 1627); 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 1627), section 5.3). We introduce the following notation:

$$\xi = m + n - p$$
$$\delta = m + n - \frac{p+q}{2}$$

$$C_3 : -\Re(b_j) < 1 \text{ for } j = 1, \ldots, m$$
$$0 < -\Re(a_j) \text{ for } j = 1, \ldots, n$$

$$C_3^* : -\Re(b_j) < 1 \text{ for } j = 1, \ldots, q$$
$$0 < -\Re(a_j) \text{ for } j = 1, \ldots, p$$

$$C_4 : -\Re(\delta) + \frac{q+1-p}{2} > q - p$$

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

   $$A = 1 \leq n \wedge p < q \wedge 1 \leq m$$

$$B = 1 \leq p \wedge 1 \leq m \wedge q = p + 1 \wedge \neg(n = 0 \wedge m = p + 1)$$

$$C = 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \ldots \left\lceil \frac{\delta}{2} \right\rceil.$$

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:

$$b^* = s + t - \frac{u+v}{2}$$

$$c^* = m + n - \frac{p+q}{2}$$

$$\rho = \sum_{j=1}^{v} d_j - \sum_{j=1}^{u} c_j + \frac{u-v}{2} + 1$$

$$\mu = \sum_{j=1}^{q} b_j - \sum_{j=1}^{p} a_j + \frac{p-q}{2} + 1$$

$$\phi = q - p - \frac{u-v}{2} + 1$$

$$\eta = 1 - (v-u) - \mu - \rho$$

$$\psi = \frac{\pi(q-m-n) + |\arg(\omega)|}{q-p}$$

$$\theta = \frac{\pi(v-s-t) + |\arg(\sigma)|)}{v-u}$$

$$\lambda_c = (q-p)|\omega|^{1/(q-p)} \cos\psi + (v-u)|\sigma|^{1/(v-u)} \cos\theta$$

$$\lambda_{s0}(c_1, c_2) = c_1(q-p)|\omega|^{1/(q-p)} \sin\psi + c_2(v-u)|\sigma|^{1/(v-u)} \sin\theta$$

$$\lambda_s = \begin{cases} \lambda_{\mathrm{s0}}\left(-1,-1\right)\lambda_{\mathrm{s0}}\left(1,1\right) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) = 0 \\ \lambda_{\mathrm{s0}}\left(\mathrm{sign}\left(\arg\left(\omega\right)\right),-1\right)\lambda_{\mathrm{s0}}\left(\mathrm{sign}\left(\arg\left(\omega\right)\right),1\right) & \text{for } \arg(\omega) \neq 0 \wedge \arg(\sigma) = 0 \\ \lambda_{\mathrm{s0}}\left(-1,\mathrm{sign}\left(\arg\left(\sigma\right)\right)\right)\lambda_{\mathrm{s0}}\left(1,\mathrm{sign}\left(\arg\left(\sigma\right)\right)\right) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) \neq 0) \\ \lambda_{\mathrm{s0}}\left(\mathrm{sign}\left(\arg\left(\omega\right)\right),\mathrm{sign}\left(\arg\left(\sigma\right)\right)\right) & \text{otherwise} \end{cases}$$

$$z_0 = \frac{\omega}{\sigma} e^{-i\pi(b^* + c^*)}$$

$$z_1 = \frac{\sigma}{\omega} e^{-i\pi(b^* + c^*)}$$

The following conditions will be helpful:

$$\begin{aligned} C_1 : &(a_i - b_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \ldots, n, j = 1, \ldots, m) \\ &\wedge (c_i - d_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \ldots, t, j = 1, \ldots, s) \end{aligned}$$

$$C_2 : \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \ldots, m, j = 1, \ldots, s$$

$$C_3 : \Re(a_i + c_j) < 1 \text{ for } i = 1, \ldots, n, j = 1, \ldots, t$$

$$C_4 : (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \ldots, t$$

$$C_5 : (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \ldots, s$$

$$C_6 : (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \ldots, n$$

$$C_7 : (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \ldots, m$$

$$C_8 : 0 < |\phi| + 2\Re\left((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)\right)$$

$$C_9 : 0 < |\phi| - 2\Re\left((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)\right)$$

$$C_{10} : |\arg(\sigma)| < \pi b^*$$

$$C_{11} : |\arg(\sigma)| = \pi b^*$$

$$C_{12} : |\arg(\omega)| < c^*\pi$$

$$C_{13} : |\arg(\omega)| = c^*\pi$$

$$C_{14}^1 : (z_0 \neq 1 \wedge |\arg(1 - z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu + \rho - u + v) < 1)$$

$$C_{14}^2 : (z_1 \neq 1 \wedge |\arg(1 - z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu + \rho - p + q) < 1)$$

$$C_{14} : \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2)$$

$$C_{15} : \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0)$$

$$C_{16} : \int_0^\infty G_{u,v}^{s,t}(\sigma x)\mathrm{d}x \text{ converges at infinity}$$

$$C_{17} : \int_0^\infty G_{p,q}^{m,n}(\omega x)\mathrm{d}x \text{ converges at infinity}$$

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. $$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re\left(\mu + \rho\right) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re\left(\mu + \rho\right) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

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.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge$$

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.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge$$

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 1627), 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

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a) \mathrm{d}z,$$

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

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s) z^s \mathrm{d}s.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} \mathrm{d}s \mathrm{d}z.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{\mathrm{d}z}{2\pi i} \mathrm{d}s.$$

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

$$f(t) = \frac{1}{t}\frac{1}{2\pi i}\int_L \Delta(s)\frac{1}{\Gamma(-as)}\left(\frac{b}{t^a}\right)^s \mathrm{d}s,$$

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 integal 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 1627), section 5.2, i.e. either $\delta >= \frac{a}{2}$ or $p \geq 1, p \geq q$.

When the integral exists

Using [Luke1969] (page 1627), 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

$$\lim_{T\to\infty}\int_{c-iT}^{c+iT} e^{zt} z^a e^{bz^c}\mathrm{d}z.$$

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write $z = c + i\tau$. Then $arg(z) \to \pm\frac{\pi}{2}$, and so $e^{zt} \sim e^{it\tau}$ (where $\sim$ shall always mean "asymptotically equivalent up to a positive real multiplicative constant"). Also $z^{x+iy} \sim |\tau|^x e^{iy\log|\tau|} e^{\pm xi\frac{\pi}{2}}$.

Set $\omega_\pm = be^{\pm i\Re(c)\frac{\pi}{2}}$. 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:

$$a = aG_{1,1}^{1,0}\left(\begin{matrix} & 1 \\ 0 & \end{matrix}\middle| z\right) + aG_{1,1}^{0,1}\left(\begin{matrix} 1 & \\ & 0 \end{matrix}\middle| z\right)$$

$$\left(z^q p + b\right)^{-a} = \frac{b^{-a}}{\Gamma\left(a\right)} G_{1,1}^{1,1}\left(\begin{matrix} -a+1 \\ 0 \end{matrix}\middle| \frac{z^q p}{b}\right)$$

$$\frac{-b^a + \left(z^q p\right)^a}{z^q p - b} = \frac{1}{\pi} b^{a-1} \sin\left(\pi a\right) G_{2,2}^{2,2}\left(\begin{matrix} 0, a \\ 0, a \end{matrix}\middle| \frac{z^q p}{b}\right)$$

$$\left(a + \sqrt{z^q p + a^2}\right)^b = -\frac{a^b b}{2\sqrt{\pi}} G_{2,2}^{1,2}\left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ 0 \end{matrix}\middle| \begin{matrix} & b \\ & \end{matrix}\middle| \frac{z^q p}{a^2}\right)$$

$$\left(-a + \sqrt{z^q p + a^2}\right)^b = \frac{a^b b}{2\sqrt{\pi}} G_{2,2}^{1,2}\left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ b \end{matrix}\middle| \begin{matrix} & 0 \\ & \end{matrix}\middle| \frac{z^q p}{a^2}\right)$$

$$\frac{\left(a + \sqrt{z^q p + a^2}\right)^b}{\sqrt{z^q p + a^2}} = \frac{1}{\sqrt{\pi}} a^{b-1} G_{2,2}^{1,2}\left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ 0 \end{matrix}\middle| \begin{matrix} & b \\ & \end{matrix}\middle| \frac{z^q p}{a^2}\right)$$

$$\frac{\left(-a + \sqrt{z^q p + a^2}\right)^b}{\sqrt{z^q p + a^2}} = \frac{1}{\sqrt{\pi}} a^{b-1} G_{2,2}^{1,2}\left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ b \end{matrix}\middle| \begin{matrix} & 0 \\ & \end{matrix}\middle| \frac{z^q p}{a^2}\right)$$

$$\left(z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a}\right)^b = -\frac{a^{\frac{b}{2}} b}{2\sqrt{\pi}} G_{2,2}^{2,1}\left(\begin{matrix} \frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix}\middle| \begin{matrix} -\frac{b}{2} + 1 \\ \end{matrix}\middle| \frac{z^q p}{a}\right)$$

$$\left(-z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a}\right)^b = \frac{a^{\frac{b}{2}} b}{2\sqrt{\pi}} G_{2,2}^{2,1}\left(\begin{matrix} -\frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix}\middle| \begin{matrix} \frac{b}{2} + 1 \\ \end{matrix}\middle| \frac{z^q p}{a}\right)$$

$$\frac{\left(z^{\frac{q}{2}}\sqrt{p}+\sqrt{z^{q}p+a}\right)^{b}}{\sqrt{z^{q}p+a}}=\frac{1}{\sqrt{\pi}}a^{\frac{b}{2}-\frac{1}{2}}G_{2,2}^{2,1}\left(\begin{matrix}\frac{b}{2}+\frac{1}{2} & -\frac{b}{2}+\frac{1}{2}\\ 0,\frac{1}{2} & \end{matrix}\middle|\frac{z^{q}p}{a}\right)$$

$$\frac{\left(-z^{\frac{q}{2}}\sqrt{p}+\sqrt{z^{q}p+a}\right)^{b}}{\sqrt{z^{q}p+a}}=\frac{1}{\sqrt{\pi}}a^{\frac{b}{2}-\frac{1}{2}}G_{2,2}^{2,1}\left(\begin{matrix}-\frac{b}{2}+\frac{1}{2} & \frac{b}{2}+\frac{1}{2}\\ 0,\frac{1}{2} & \end{matrix}\middle|\frac{z^{q}p}{a}\right)$$

Functions involving $|-z^{q}p+b|$:

$$|-z^{q}p+b|^{-a}=\frac{\pi\left|b\right|^{-a}}{\cos\left(\frac{\pi a}{2}\right)\Gamma\left(a\right)}G_{2,2}^{1,1}\left(\begin{matrix}-a+1 & -\frac{a}{2}+\frac{1}{2}\\ 0 & -\frac{a}{2}+\frac{1}{2}\end{matrix}\middle|\frac{z^{q}p}{b}\right),\ \text{if}\ \Re a<1$$

Functions involving $\operatorname{Chi}\left(z^{q}p\right)$:

$$\operatorname{Chi}\left(z^{q}p\right)=-\frac{\pi^{\frac{3}{2}}}{2}G_{2,4}^{2,0}\left(\begin{matrix} & \frac{1}{2},1\\ 0,0 & \frac{1}{2},\frac{1}{2}\end{matrix}\middle|\frac{p^{2}}{4}z^{2q}\right)$$

Functions involving $\operatorname{Ci}\left(z^{q}p\right)$:

$$\operatorname{Ci}\left(z^{q}p\right)=-\frac{\sqrt{\pi}}{2}G_{1,3}^{2,0}\left(\begin{matrix} & 1\\ 0,0 & \frac{1}{2}\end{matrix}\middle|\frac{p^{2}}{4}z^{2q}\right)$$

Functions involving $\operatorname{Ei}\left(z^{q}p\right)$:

$$\operatorname{Ei}\left(z^{q}p\right)=-i\pi G_{1,1}^{1,0}\left(\begin{matrix} & 1\\ 0 & \end{matrix}\middle|z\right)-G_{1,2}^{2,0}\left(\begin{matrix} & 1\\ 0,0 & \end{matrix}\middle|z^{q}pe^{i\pi}\right)-i\pi G_{1,1}^{0,1}\left(\begin{matrix}1 & \\ & 0\end{matrix}\middle|z\right)$$

Functions involving $\theta\left(z^{q}p-b\right)$:

$$\left(z^{q}p-b\right)^{a-1}\theta\left(z^{q}p-b\right)=b^{a-1}\Gamma\left(a\right)G_{1,1}^{0,1}\left(\begin{matrix}a & \\ & 0\end{matrix}\middle|\frac{z^{q}p}{b}\right),\ \text{if}\ b>0$$

$$\left(-z^{q}p+b\right)^{a-1}\theta\left(-z^{q}p+b\right)=b^{a-1}\Gamma\left(a\right)G_{1,1}^{1,0}\left(\begin{matrix} & a\\ 0 & \end{matrix}\middle|\frac{z^{q}p}{b}\right),\ \text{if}\ b>0$$

$$\left(z^{q}p-b\right)^{a-1}\theta\left(z-\left(\frac{b}{p}\right)^{\frac{1}{q}}\right)=b^{a-1}\Gamma\left(a\right)G_{1,1}^{0,1}\left(\begin{matrix}a & \\ & 0\end{matrix}\middle|\frac{z^{q}p}{b}\right),\ \text{if}\ b>0$$

$$\left(-z^{q}p+b\right)^{a-1}\theta\left(-z+\left(\frac{b}{p}\right)^{\frac{1}{q}}\right)=b^{a-1}\Gamma\left(a\right)G_{1,1}^{1,0}\left(\begin{matrix} & a\\ 0 & \end{matrix}\middle|\frac{z^{q}p}{b}\right),\ \text{if}\ b>0$$

Functions involving $\theta\left(-z^q p+1\right)$, $\log\left(z^q p\right)$:

$$\log^n\left(z^q p\right)\theta\left(-z^q p+1\right) = \text{generated}$$

$$\log^n\left(z^q p\right)\theta\left(z^q p-1\right) = \text{generated}$$

Functions involving $\operatorname{Shi}\left(z^q p\right)$:

$$\operatorname{Shi}\left(z^q p\right) = \frac{\sqrt{\pi}p}{4} z^q G_{1,3}^{1,1}\left(\begin{matrix}\frac{1}{2} \\ 0\end{matrix}\quad \begin{matrix} \\ -\frac{1}{2}, -\frac{1}{2}\end{matrix}\middle| \frac{p^2}{4} z^{2q} e^{i\pi}\right)$$

Functions involving $\operatorname{Si}\left(z^q p\right)$:

$$\operatorname{Si}\left(z^q p\right) = \frac{\sqrt{\pi}}{2} G_{1,3}^{1,1}\left(\begin{matrix}1 \\ \frac{1}{2}\end{matrix}\quad \begin{matrix} \\ 0, 0\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $I_a\left(z^q p\right)$:

$$I_a\left(z^q p\right) = \pi G_{1,3}^{1,0}\left(\begin{matrix} \\ \frac{a}{2}\end{matrix}\quad \begin{matrix}\frac{a}{2}+\frac{1}{2} \\ -\frac{a}{2}, \frac{a}{2}+\frac{1}{2}\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $J_a\left(z^q p\right)$:

$$J_a\left(z^q p\right) = G_{0,2}^{1,0}\left(\begin{matrix} \\ \frac{a}{2}\end{matrix}\quad \begin{matrix} \\ -\frac{a}{2}\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $K_a\left(z^q p\right)$:

$$K_a\left(z^q p\right) = \frac{1}{2} G_{0,2}^{2,0}\left(\begin{matrix} \\ \frac{a}{2}, -\frac{a}{2}\end{matrix}\quad \begin{matrix} \\ \end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $Y_a\left(z^q p\right)$:

$$Y_a\left(z^q p\right) = G_{1,3}^{2,0}\left(\begin{matrix} \\ \frac{a}{2}, -\frac{a}{2}\end{matrix}\quad \begin{matrix}-\frac{a}{2}-\frac{1}{2} \\ -\frac{a}{2}-\frac{1}{2}\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $\cos\left(z^q p\right)$:

$$\cos\left(z^q p\right) = \sqrt{\pi} G_{0,2}^{1,0}\left(\begin{matrix} \\ 0\end{matrix}\quad \begin{matrix} \\ \frac{1}{2}\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $\cosh\left(z^q p\right)$:

$$\cosh\left(z^q p\right) = \pi^{\frac{3}{2}} G_{1,3}^{1,0}\left(\begin{matrix} \\ 0\end{matrix}\quad \begin{matrix}\frac{1}{2} \\ \frac{1}{2}, \frac{1}{2}\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $E\left(z^q p\right)$:

$$E\left(z^q p\right) = -\frac{1}{4} G_{2,2}^{1,2}\left(\begin{matrix} \frac{1}{2}, \frac{3}{2} & \\ 0 & 0 \end{matrix}\middle| -z^q p\right)$$

Functions involving $K\left(z^q p\right)$:

$$K\left(z^q p\right) = \frac{1}{2} G_{2,2}^{1,2}\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} & \\ 0 & 0 \end{matrix}\middle| -z^q p\right)$$

Functions involving $\operatorname{erf}\left(z^q p\right)$:

$$\operatorname{erf}\left(z^q p\right) = \frac{1}{\sqrt{\pi}} G_{1,2}^{1,1}\left(\begin{matrix} 1 & \\ \frac{1}{2} & 0 \end{matrix}\middle| z^{2q} p^2\right)$$

Functions involving $\operatorname{erfc}\left(z^q p\right)$:

$$\operatorname{erfc}\left(z^q p\right) = \frac{1}{\sqrt{\pi}} G_{1,2}^{2,0}\left(\begin{matrix} & 1 \\ 0, \frac{1}{2} & \end{matrix}\middle| z^{2q} p^2\right)$$

Functions involving $\operatorname{erfi}\left(z^q p\right)$:

$$\operatorname{erfi}\left(z^q p\right) = \frac{z^q p}{\sqrt{\pi}} G_{1,2}^{1,1}\left(\begin{matrix} \frac{1}{2} & \\ 0 & -\frac{1}{2} \end{matrix}\middle| -z^{2q} p^2\right)$$

Functions involving $e^{z^q p e^{i\pi}}$:

$$e^{z^q p e^{i\pi}} = G_{0,1}^{1,0}\left(\begin{matrix} & \\ 0 & \end{matrix}\middle| z^q p\right)$$

Functions involving $\mathrm{E}_a\left(z^q p\right)$:

$$\mathrm{E}_a\left(z^q p\right) = G_{1,2}^{2,0}\left(\begin{matrix} & a \\ a-1, 0 & \end{matrix}\middle| z^q p\right)$$

Functions involving $C\left(z^q p\right)$:

$$C\left(z^q p\right) = \frac{1}{2} G_{1,3}^{1,1}\left(\begin{matrix} 1 & \\ \frac{1}{4} & 0, \frac{3}{4} \end{matrix}\middle| \frac{\pi^2 p^4}{16} z^{4q}\right)$$

Functions involving $S\left(z^q p\right)$:

$$S\left(z^q p\right) = \frac{1}{2} G_{1,3}^{1,1}\left(\begin{matrix} 1 & \\ \frac{3}{4} & 0, \frac{1}{4} \end{matrix}\middle| \frac{\pi^2 p^4}{16} z^{4q}\right)$$

Functions involving $\log\left(z^q p\right)$:

$$\log^n\left(z^q p\right) = \text{generated}$$

$$\log\left(z^q p + a\right) = \log\left(a\right) G_{1,1}^{1,0}\left(\begin{matrix} & 1 \\ 0 & \end{matrix}\middle| z\right) + \log\left(a\right) G_{1,1}^{0,1}\left(\begin{matrix} 1 & \\ & 0\end{matrix}\middle| z\right) + G_{2,2}^{1,2}\left(\begin{matrix} 1, 1 & \\ 1 & 0\end{matrix}\middle| \frac{z^q p}{a}\right)$$

$$\log\left(\left|z^q p - a\right|\right) = \log\left(\left|a\right|\right) G_{1,1}^{1,0}\left(\begin{matrix} & 1 \\ 0 & \end{matrix}\middle| z\right) + \log\left(\left|a\right|\right) G_{1,1}^{0,1}\left(\begin{matrix} 1 & \\ & 0\end{matrix}\middle| z\right) + \pi G_{3,3}^{1,2}\left(\begin{matrix} 1, 1 & \frac{1}{2} \\ 1 & 0, \frac{1}{2}\end{matrix}\middle| \frac{z^q p}{a}\right)$$

Functions involving $\sin\left(z^q p\right)$:

$$\sin\left(z^q p\right) = \sqrt{\pi} G_{0,2}^{1,0}\left(\begin{matrix} & \\ \frac{1}{2} & 0\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

Functions involving $\sinh\left(z^q p\right)$:

$$\sinh\left(z^q p\right) = \pi^{\frac{3}{2}} G_{1,3}^{1,0}\left(\begin{matrix} & 1 \\ \frac{1}{2} & 1, 0\end{matrix}\middle| \frac{p^2}{4} z^{2q}\right)$$

5.11.4 API reference

static `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 +-oo, 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`, `Integral.doit`

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)
```

static integrals.**line_integrate**(*field, Curve, variables*)

Compute the line integral.

See Also:

`integrate`, `Integral`

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)
```

static integrals.**deltaintegrate**(*f*, *x*)

The idea for integration is the following:

- If we are dealing with a DiracDelta expression, i.e. DiracDelta(g(x)), we try to simplify it.

  If we could simplify it, then we integrate the resulting expression. We already know we can integrate a simplified expression, because only simple DiracDelta expressions are involved.

  If we couldn't simplify it, there are two cases:

  1. The expression is a simple expression: we return the integral, taking care if we are dealing with a Derivative or with a proper DiracDelta.
  2. The expression is not simple (i.e. DiracDelta(cos(x))): we can do nothing at all.

- If the node is a multiplication node having a DiracDelta term:

  First we expand it.

  If the expansion did work, the we try to integrate the expansion.

  If not, we try to extract a simple DiracDelta term, then we have two cases:

  1. We have a simple DiracDelta term, so we return the integral.
  2. We didn't have a simple term, but we do have an expression with simplified DiracDelta terms, so we integrate this expression.

See Also:

sympy.functions.special.delta_functions.DiracDelta (page 336), sympy.integrals.integrals.Integral

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.integrals.deltafunctions import deltaintegrate
>>> from sympy import sin, cos, DiracDelta, Heaviside
>>> deltaintegrate(x*sin(x)*cos(x)*DiracDelta(x - 1), x)
sin(1)*cos(1)*Heaviside(x - 1)
>>> deltaintegrate(y**2*DiracDelta(x - z)*DiracDelta(y - z), y)
z**2*DiracDelta(x - z)*Heaviside(y - z)
```

static integrals.**ratint**(*f*, *x*, ***flags*)

Performs indefinite integration of rational functions.

Given a field K and a rational function $f = p/q$, where p and q are polynomials in $K[x]$, returns a function g such that $f = g'$.

```
>>> from sympy.integrals.rationaltools import ratint
>>> from sympy.abc import x

>>> ratint(36/(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2), x)
(12*x + 6)/(x**2 - 1) + 4*log(x - 2) - 4*log(x + 1)
```

See Also:

sympy.integrals.integrals.Integral.doit, ratint_logpart, ratint_ratpart

References

[Bro05] (page 1624)

static `integrals.`**`heurisch`**(*f, x, rewrite=False, hints=None, mappings=None, retries=3, degree_offset=0, unnecessary_permutations=None*)

Compute indefinite integral using heuristic Risch algorithm.

This is a heuristic approach to indefinite integration in finite terms using the extended heuristic (parallel) Risch algorithm, based on Manuel Bronstein's "Poor Man's Integrator".

The algorithm supports various classes of functions including transcendental elementary or special functions like Airy, Bessel, Whittaker and Lambert.

Note that this algorithm is not a decision procedure. If it isn't able to compute the antiderivative for a given function, then this is not a proof that such a functions does not exist. One should use recursive Risch algorithm in such case. It's an open question if this algorithm can be made a full decision procedure.

This is an internal integrator procedure. You should use toplevel 'integrate' function in most cases, as this procedure needs some preprocessing steps and otherwise may fail.

See Also:

`sympy.integrals.integrals.Integral.doit`, `sympy.integrals.integrals.Integral`, `components`

Examples

```
>>> from sympy import tan
>>> from sympy.integrals.heurisch import heurisch
>>> from sympy.abc import x, y

>>> heurisch(y*tan(x), x)
y*log(tan(x)**2 + 1)/2
```

See Manuel Bronstein's "Poor Man's Integrator":

[1] http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint/index.html

For more information on the implemented algorithm refer to:

[2] K. Geddes, L. Stefanus, On the Risch-Norman Integration Method and its Implementation in Maple, Proceedings of ISSAC'89, ACM Press, 212-217.

[3] J. H. Davenport, On the Parallel Risch Algorithm (I), Proceedings of EUROCAM'82, LNCS 144, Springer, 144-157.

[4] J. H. Davenport, On the Parallel Risch Algorithm (III): Use of Tangents, SIGSAM Bulletin 16 (1982), 3-6.

[5] J. H. Davenport, B. M. Trager, On the Parallel Risch Algorithm (II), ACM Transactions on Mathematical Software 11 (1985), 356-362.

Specification

heurisch(f, x, rewrite=False, hints=None)

where f : expression x : symbol

rewrite -> force rewrite 'f' in terms of 'tan' and 'tanh' hints -> a list of functions that may appear in anti-derivate

- hints = None –> no suggestions at all
- hints = [ ] –> try to figure out
- hints = [f1, ..., fn] –> we know better

static integrals.**heurisch_wrapper**(*f, x, rewrite=False, hints=None, mappings=None, retries=3, degree_offset=0, unnecessary_permutations=None*)

A wrapper around the heurisch integration algorithm.

This method takes the result from heurisch and checks for poles in the denominator. For each of these poles, the integral is reevaluated, and the final integration result is given in terms of a Piecewise.

See Also:

heurisch

Examples

```
>>> from sympy.core import symbols
>>> from sympy.functions import cos
>>> from sympy.integrals.heurisch import heurisch, heurisch_wrapper
>>> n, x = symbols('n x')
>>> heurisch(cos(n*x), x)
sin(n*x)/n
>>> heurisch_wrapper(cos(n*x), x)
Piecewise((x, n == 0), (sin(n*x)/n, True))
```

static integrals.**trigintegrate**(*f, x, conds='piecewise'*)

Integrate f = Mul(trig) over x

```
>>> from sympy import Symbol, sin, cos, tan, sec, csc, cot
>>> from sympy.integrals.trigonometry import trigintegrate
>>> from sympy.abc import x

>>> trigintegrate(sin(x)*cos(x), x)
sin(x)**2/2

>>> trigintegrate(sin(x)**2, x)
x/2 - sin(x)*cos(x)/2

>>> trigintegrate(tan(x)*sec(x), x)
1/cos(x)

>>> trigintegrate(sin(x)*tan(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2 - sin(x)
```

http://en.wikibooks.org/wiki/Calculus/Integration_techniques

See Also:

sympy.integrals.integrals.Integral.doit, sympy.integrals.integrals.Integral

static integrals.**manualintegrate**(*f*, *var*)

Compute indefinite integral of a single variable using an algorithm that resembles what a student would do by hand.

Unlike `integrate`, var can only be a single symbol.

See Also:

`sympy.integrals.integrals.integrate`, `sympy.integrals.integrals.Integral.doit`, `sympy.integrals.integrals.Integral`

Examples

```
>>> from sympy import sin, cos, tan, exp, log, integrate
>>> from sympy.integrals.manualintegrate import manualintegrate
>>> from sympy.abc import x
>>> manualintegrate(1 / x, x)
log(x)
>>> integrate(1/x)
log(x)
>>> manualintegrate(log(x), x)
x*log(x) - x
>>> integrate(log(x))
x*log(x) - x
>>> manualintegrate(exp(x) / (1 + exp(2 * x)), x)
atan(exp(x))
>>> integrate(exp(x) / (1 + exp(2 * x)))
RootSum(4*_z**2 + 1, Lambda(_i, _i*log(2*_i + exp(x))))
>>> manualintegrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> manualintegrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> manualintegrate(tan(x), x)
-log(cos(x))
>>> integrate(tan(x), x)
-log(sin(x)**2 - 1)/2
```

static manualintegrate.**integral_steps**(*integrand*, *symbol*, ***options*)

Returns the steps needed to compute an integral.

This function attempts to mirror what a student would do by hand as closely as possible.

Returns **rule** : namedtuple

The first step; most rules have substeps that must also be considered. These substeps can be evaluated using *manualintegrate* to obtain a result.

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*, *method='midpoint'*)

Approximates the definite integral by a sum.

method ... one of: left, right, midpoint, trapezoid

These are all basically the rectangle method [1], the only difference is where the function value is taken in each interval to define the rectangle.

[1] http://en.wikipedia.org/wiki/Rectangle_method

See Also:

Integral.doit Perform the integration using any hints

Examples

```
>>> from sympy import sin, sqrt
>>> from sympy.abc import x
>>> from sympy.integrals import Integral
>>> e = Integral(sin(x), (x, 3, 7))
>>> 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
```

All but the trapexoid method may be used when dealing with a function with a discontinuity. 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 oo:

```
>>> e.as_sum(5, 'left')
oo
>>> e.as_sum(5, 'trapezoid')
oo
```

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 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((2, log(x) == 0), (x**3/log(x) - x/log(x), 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
set([y])
```

function

Return the function to be integrated.

See Also:

`limits`, `variables`, `free_symbols`

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x
>>> Integral(x**2, (x,)).function
x**2
```

is_number

Return True if the Integral will result in a number, else False.

Integrals are a special case since they contain symbols that can be replaced with numbers. Whether the integral can be done or not is another issue. But answering whether the final result is a number is not difficult.

See Also:

`is_zero`

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x).is_number
False
>>> Integral(x, y).is_number
False
>>> Integral(x, (y, 1, x)).is_number
False
>>> Integral(x, (y, 1, 2)).is_number
False
>>> Integral(x, (y, 1, 1)).is_number
True
>>> Integral(x, (x, 1, 2)).is_number
True
>>> Integral(x*y, (x, 1, 2), (y, 1, 3)).is_number
True
>>> Integral(1, x, (x, 1, 2)).is_number
True
```

is_zero

Since Integral doesn't autosimplify it it useful to see if it would simplify to zero or not in a trivial manner, i.e. when the function is 0 or two limits of a definite integral are the same.

This is a very naive and quick test, not intended to check for special patterns like Integral(sin(m*x)*cos(n*x), (x, 0, 2*pi)) == 0.

See Also:

`is_number`

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, y, z
>>> Integral(1, (x, 1, 1)).is_zero
True
>>> Integral(0, (x, y, z)).is_zero
True
>>> Integral(1, (x, 1, 2)).is_zero
False
```

limits

Return the limits of integration.

See Also:

`function`, `variables`, `free_symbols`

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)
```

transform(*x*, *u*, *inverse=False*)

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.

The $inverse$ option will reverse x and u. It is a deprecated option since x and u can just be passed in reverse order.

Once f and F have been identified, the transformation is made as follows:

$$\int_a^b x\mathrm{d}x \rightarrow \int_{F(a)}^{F(b)} f(x)\frac{\mathrm{d}}{\mathrm{d}x}$$

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 $sqrt(x)$, 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 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 = +/-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))
```

variables

Return a list of the integration variables.

See Also:

`function`, `limits`, `free_symbols`

as_dummy Replace integration variables with dummy ones

transform Perform mapping on the integration variable

Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).variables
[i]
```

5.11.5 TODO and Bugs

There are still lots of functions that SymPy does not know how to integrate. For bugs related to this module, see http://code.google.com/p/sympy/issues/list?q=label:Integration

5.12 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 [R189] (page 1624) 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:

$$w_i = \frac{2}{(1-x_i^2)\left(P_n'(x_i)\right)^2}$$

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`, `gauss_gen_laguerre`, `gauss_hermite`, `gauss_chebyshev_t`, `gauss_chebyshev_u`, `gauss_jacobi`

References

[R189] (page 1624), [R190] (page 1624)

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
[0.34786, 0.65215, 0.65215, 0.34786]
```

sympy.integrals.quadrature.**gauss_laguerre**(*n, n_digits*)

Computes the Gauss-Laguerre quadrature [R191] (page 1624) 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:

$$w_i = \frac{x_i}{(n+1)^2 \left(L_{n+1}(x_i)\right)^2}$$

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`, `gauss_gen_laguerre`, `gauss_hermite`, `gauss_chebyshev_t`, `gauss_chebyshev_u`, `gauss_jacobi`

References

[R191] (page 1624), [R192] (page 1624)

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 [R193] (page 1624) 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:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 \left(H_{n-1}(x_i)\right)^2}$$

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`, `gauss_laguerre`, `gauss_gen_laguerre`, `gauss_chebyshev_t`, `gauss_chebyshev_u`, `gauss_jacobi`

References

[R193] (page 1624), [R194] (page 1624), [R195] (page 1624)

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 [R196] (page 1624) 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`, `gauss_laguerre`, `gauss_hermite`, `gauss_chebyshev_t`, `gauss_chebyshev_u`, `gauss_jacobi`

References

[R196] (page 1624), [R197] (page 1625)

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 [R198] (page 1625) 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`, `gauss_laguerre`, `gauss_hermite`, `gauss_gen_laguerre`, `gauss_chebyshev_u`, `gauss_jacobi`

References

[R198] (page 1625), [R199] (page 1625)

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 [R200] (page 1625) 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, gauss_laguerre, gauss_hermite, gauss_gen_laguerre, gauss_chebyshev_t, gauss_jacobi

References

[R200] (page 1625), [R201] (page 1625)

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 [R202] (page 1625) 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`, `gauss_laguerre`, `gauss_hermite`, `gauss_gen_laguerre`, `gauss_chebyshev_t`, `gauss_chebyshev_u`

References

[R202] (page 1625), [R203] (page 1625), [R204] (page 1625)

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]
```

5.13 Logic Module

5.13.1 Introduction

The logic module for SymPy allows to form and manipulate logic expressions using symbolic and Boolean values.

5.13.2 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)
Or(And(x, y), y)
>>> x | y
Or(x, y)
>>> ~x
Not(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()
set([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

[R205] (page 1625)

Examples

```
>>> from sympy.logic import SOPform
>>> 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)
Or(And(Not(w), z), And(y, z))
```

`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

[R206] (page 1625)

Examples

```
>>> from sympy.logic import POSform
>>> 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)
And(Or(Not(w), y), z)
```

5.13.3 Boolean functions

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 operator & will perform bitwise operations on integers and for convenience will construct an Add when the arguments are symbolic, but the And function does not perform bitwise operations and when any argument is True it is simply removed from the arguments:

```
>>> 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
And(x, y)
```

Attributes

nargs	

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 operator | will perform bitwise operations on integers and for convenience will construct an Or when the arguments are symbolic, but the Or function does not

perform bitwise operations and when any argument is False it is simply removed from the arguments:

```
>>> 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
Or(x, y)
```

Attributes

nargs	

class sympy.logic.boolalg.**Not**

Logical Not function (negation)

Notes

De Morgan rules are applied automatically.

Attributes

nargs	

class sympy.logic.boolalg.**Xor**

Logical XOR (exclusive OR) function.

Attributes

nargs	

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.

Attributes

nargs	

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.

Attributes

nargs	

class sympy.logic.boolalg.**Implies**

Logical implication.

A implies B is equivalent to !A v B

Attributes

nargs	

class sympy.logic.boolalg.**Equivalent**

Equivalence relation.

Equivalent(A, B) is True iff A and B are both True or both False

Attributes

nargs	

class sympy.logic.boolalg.**ITE**

If then else clause.

Attributes

nargs	

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 | ~B | ...) & (B | C | ...) & ...)

Examples

```
>>> from sympy.logic.boolalg import to_cnf
>>> from sympy.abc import A, B, D
>>> to_cnf(~(A | B) | D)
And(Or(D, Not(A)), Or(D, Not(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 & ~B & ...) | (B & C & ...) | ...)

Examples

```
>>> from sympy.logic.boolalg import to_dnf
>>> from sympy.abc import A, B, C, D
>>> to_dnf(B & (A | C))
Or(And(A, B), And(B, 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
```

5.13.4 Simplification and equivalence-testing

sympy.logic.boolalg.**simplify_logic**(*expr*)

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. The input can be a string or a boolean expression.

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)
And(Not(x), Not(y))

>>> S(b)
Or(And(Not(x), Not(y), Not(z)), And(Not(x), Not(y), z))
>>> simplify_logic(_)
And(Not(x), Not(y))
```

SymPy's simplify() function can also be used to simplify logic expressions to their simplest forms.

5.13.5 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 & ~x` is not satisfiable, since there are no values for `x` that make this sentence `True`. On the other hand, `(x | y) & (x | ~y) & (~x | 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'*)

Check satisfiability of a propositional sentence. Returns a model when it succeeds

Examples:

```
>>> from sympy.abc import A, B
>>> from sympy.logic.inference import satisfiable
>>> satisfiable(A & ~B)
{A: True, B: False}
>>> satisfiable(A & ~A)
False
```

5.14 Matrices

A module that handles matrices.

Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

Contents:

5.14.1 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]
>>> Matrix([M, (0, 0, -1)])
[1  0  0 ]
[        ]
[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]
[          ]
[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]
[    ]
[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]
```

```
[               ]
[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,lambda i,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]
[       ]
[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]
```

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
[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()
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 ]
[                 ]
[-1/28  5/28  -1/4]
[                 ]
[ 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
[  ___      ___      ___ ]
[\/ 6   -\/ 3    -\/ 2  ]
[-----  -------  -------]
[  6       3        2   ]
[                       ]
[  ___      ___      ___ ]
[\/ 6   -\/ 3     \/ 2  ]
[-----  -------   ----- ]
[  6       3        2   ]
[                       ]
[  ___      ___          ]
[\/ 6    \/ 3           ]
[-----   -----      0   ]
[  3       3            ]
>>> R
[             ___       ]
[  ___   4*\/ 6       ___]
[\/ 6   -------  2*\/ 6 ]
[          3            ]
[                       ]
[             ___       ]
[         \/ 3          ]
[  0      -----      0  ]
[          3            ]
[                       ]
[                   ___ ]
```

```
[  0       0      \/ 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 then with respect to another 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**

Attributes

is_Identity	

C

By-element conjugation.

D

Return Dirac conjugate (if self.rows == 4).

See Also:

conjugate By-element conjugation

H **(page 556)** 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.
```

H

Return Hermite conjugate.

See Also:

conjugate By-element conjugation

D (page 556) Dirac conjugation

Examples

```
>>> from sympy import Matrix, I
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([
[    0],
[1 + I],
[    2],
[    3]])
>>> m.H
Matrix([[0, 1 - I, 2, 3]])
```

LDLdecomposition()

Returns the LDL Decomposition (L, D) of matrix A, such that L * D * L.T == 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 square, symmetric, positive-definite and non-singular matrix.

See Also:

cholesky (page 562), LUdecomposition (page 558), QRdecomposition (page 559)

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
```

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 557), lower_triangular_solve (page 578), upper_triangular_solve (page 586), cholesky_solve (page 563), diagonal_solve (page 565), LUsolve (page 558), QRsolve (page 559)

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 at 0x0000000004752E48>*)

Returns the decomposition LU and the row swaps p.

See Also:

cholesky (page 562), LDLdecomposition (page 557), QRdecomposition (page 559), LUdecomposition_Simple (page 558), LUdecompositionFF (page 558), LUsolve (page 558)

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 558), LUdecomposition_Simple (page 558), LUsolve (page 558)

LUdecomposition_Simple(*iszerofunc=<function _iszero at 0x0000000004752E48>*)

Returns A comprised of L, U (L's diag entries are 1) and p which is the list of the row swaps (in order).

See Also:

LUdecomposition (page 558), LUdecompositionFF (page 558), LUsolve (page 558)

LUsolve(*rhs, iszerofunc=<function _iszero at 0x0000000004752E48>*)

Solve the linear system Ax = rhs for x where A = self.

This is for symbolic matrices, for real or complex ones use sympy.mpmath.lu_solve or sympy.mpmath.qr_solve.

See Also:

lower_triangular_solve (page 578), upper_triangular_solve (page 586), cholesky_solve (page 563), diagonal_solve (page 565), LDLsolve (page 557), QRsolve (page 559), LUdecomposition (page 558)

QRdecomposition()

Return Q, R where A = Q*R, Q is orthogonal and R is upper triangular.

See Also:

cholesky (page 562), LDLdecomposition (page 557), LUdecomposition (page 558), QRsolve (page 559)

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 sympy.mpmath.qr_solve.

See Also:

lower_triangular_solve (page 578), upper_triangular_solve (page 586), cholesky_solve (page 563), diagonal_solve (page 565), LDLsolve (page 557), LUsolve (page 558), QRdecomposition (page 559)

T

Matrix transposition.

add(*b*)

Return self + b

adjoint()

Conjugate transpose or Hermitian conjugation.

adjugate(*method='berkowitz'*)

Returns the adjugate matrix.

Adjugate matrix is the transpose of the cofactor matrix.

http://en.wikipedia.org/wiki/Adjugate

See Also:

cofactorMatrix (page 563), transpose, berkowitz (page 560)

berkowitz()

The Berkowitz algorithm.

Given N x N matrix with symbolic content, compute efficiently coefficients of characteristic polynomials of 'self' and all its square sub-matrices composed by removing both i-th row and column, without division in the ground domain.

This method is particularly useful for computing determinant, principal minors and characteristic polynomial, when 'self' has complicated coefficients e.g. polynomials. Semi-direct usage of this algorithm is also important in computing efficiently sub-resultant PRS.

Assuming that M is a square matrix of dimension N x N and I is N x N identity matrix, then the following following definition of characteristic polynomial is begin used:

charpoly(M) = det(t*I - M)

As a consequence, all polynomials generated by Berkowitz algorithm are monic.

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z

>>> M = Matrix([[x, y, z], [1, 0, 0], [y, z, x]])

>>> p, q, r = M.berkowitz()
```

```
>>> p # 1 x 1 M's sub-matrix
(1, -x)

>>> q # 2 x 2 M's sub-matrix
(1, -x, -y)

>>> r # 3 x 3 M's sub-matrix
(1, -2*x, x**2 - y*z - y, x*y - z**2)
```

For more information on the implemented algorithm refer to:

[1] S.J. Berkowitz, On computing the determinant in small parallel time using a small number of processors, ACM, Information Processing Letters 18, 1984, pp. 147-150

[2] M. Keber, Division-Free computation of sub-resultants using Bezout matrices, Tech. Report MPI-I-2006-1-006, Saarbrucken, 2006

See Also:

berkowitz_det (page 561), berkowitz_minors (page 562), berkowitz_charpoly (page 561), berkowitz_eigenvals (page 562)

berkowitz_charpoly(*x=_lambda, simplify=<function simplify at 0x00000000046833C8>*)

Computes characteristic polynomial minors using Berkowitz method.

A PurePoly is returned so using different variables for x does not affect the comparison or the polynomials:

See Also:

berkowitz (page 560)

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> A = Matrix([[1, 3], [2, 0]])
>>> A.berkowitz_charpoly(x) == A.berkowitz_charpoly(y)
True
```

Specifying x is optional; a Dummy with name lambda is used by default (which looks good when pretty-printed in unicode):

```
>>> A.berkowitz_charpoly().as_expr()
_lambda**2 - _lambda - 6
```

No test is done to see that x doesn't clash with an existing symbol, so using the default (lambda) or your own Dummy symbol is the safest option:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
_lambda**2 - _lambda - 2*x
>>> A.charpoly(x).as_expr()
x**2 - 3*x
```

berkowitz_det()
Computes determinant using Berkowitz method.

See Also:

det (page 564), berkowitz (page 560)

berkowitz_eigenvals(***flags*)
Computes eigenvalues of a Matrix using Berkowitz method.

See Also:

berkowitz (page 560)

berkowitz_minors()
Computes principal minors using Berkowitz method.

See Also:

berkowitz (page 560)

charpoly(*x=_lambda, simplify=<function simplify at 0x00000000046833C8>*)
Computes characteristic polynomial minors using Berkowitz method.

A PurePoly is returned so using different variables for x does not affect the comparison or the polynomials:

See Also:

berkowitz (page 560)

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> A = Matrix([[1, 3], [2, 0]])
>>> A.berkowitz_charpoly(x) == A.berkowitz_charpoly(y)
True
```

Specifying x is optional; a Dummy with name lambda is used by default (which looks good when pretty-printed in unicode):

```
>>> A.berkowitz_charpoly().as_expr()
_lambda**2 - _lambda - 6
```

No test is done to see that x doesn't clash with an existing symbol, so using the default (lambda) or your own Dummy symbol is the safest option:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
_lambda**2 - _lambda - 2*x
>>> A.charpoly(x).as_expr()
x**2 - 3*x
```

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.

See Also:

LDLdecomposition (page 557), LUdecomposition (page 558), QRdecomposition (page 559)

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]])
```

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 578), upper_triangular_solve (page 586), diagonal_solve (page 565), LDLsolve (page 557), LUsolve (page 558), QRsolve (page 559)

cofactor(*i*, *j*, *method='berkowitz'*)

Calculate the cofactor of an element.

See Also:

cofactorMatrix (page 563), minorEntry (page 579), minorMatrix (page 579)

cofactorMatrix(*method='berkowitz'*)

Return a matrix containing the cofactor of each element.

See Also:

cofactor (page 563), minorEntry (page 579), minorMatrix (page 579), adjugate (page 560)

col_insert(*pos*, *mti*)

Insert one or more columns at the given column position.

See Also:

col, row_insert (page 582)

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.col_insert(1, V)
Matrix([
[0, 1, 0, 0],
[0, 1, 0, 0],
[0, 1, 0, 0]])
```

col_join(*bott*)

Concatenates two matrices along self's last and bott's first row

See Also:

col, row_join (page 583)

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[1, 1, 1]])
```

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 (page 584)

Examples

```
>>> from sympy import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

cross(*b*)

Calculate the cross product of self and b.

See Also:

dot (page 566), multiply (page 579), multiply_elementwise (page 579)

det(*method='bareis'*)

Computes the matrix determinant using the method "method".

Possible values for "method": bareis ... det_bareis berkowitz ... berkowitz_det lu_decomposition ... det_LU

See Also:

det_bareis (page 564), berkowitz_det (page 561), det_LU

det_LU_decomposition()

Compute matrix determinant using LU decomposition

Note that this method fails if the LU decomposition itself fails. In particular, if the matrix has no inverse this method will fail.

TODO: Implement algorithm for sparse matrices (SFF), http://www.eecis.udel.edu/~saunders/papers/sffge/it5.ps.

See Also:

det (page 564), det_bareis (page 564), berkowitz_det (page 561)

det_bareis()

Compute matrix determinant using Bareis' fraction-free algorithm which is an extension of the well known Gaussian elimination method. This approach is best suited for dense symbolic matrices and will result in a determinant with minimal number of fractions. It means that less term rewriting is needed on resulting formulae.

TODO: Implement algorithm for sparse matrices (SFF), http://www.eecis.udel.edu/~saunders/papers/sffge/it5.ps.

See Also:

det (page 564), berkowitz_det (page 561)

diagonal_solve(*rhs*)

Solves Ax = B efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

See Also:

lower_triangular_solve (page 578), upper_triangular_solve (page 586), cholesky_solve (page 563), LDLsolve (page 557), LUsolve (page 558), QRsolve (page 559)

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True
```

diagonalize(*reals_only=False, sort=False, normalize=False*)

Return (P, D), where D is diagonal and

D = P^-1 * M * P

where M is current matrix.

See Also:

is_diagonal (page 570), is_diagonalizable (page 571)

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
Matrix([
[1,  2, 0],
[0,  3, 0],
[2, -4, 2]])
>>> (P, D) = m.diagonalize()
>>> D
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> P
Matrix([
```

```
[-1, 0, -1],
[ 0, 0, -1],
[ 2, 1,  2]])
>>> P.inv() * m * P
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

diff(**args*)

Calculate the derivative of each element in the matrix.

See Also:

integrate (page 569), limit (page 578)

Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

dot(*b*)

Return the dot product of Matrix self and b relaxing the condition of compatible dimensions: if either the number of rows or columns are the same as the length of b then the dot product is returned. If self is a row or column vector, a scalar is returned. Otherwise, a list of results is returned (and in that case the number of columns in self must match the length of b).

See Also:

cross (page 564), multiply (page 579), multiply_elementwise (page 579)

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = [1, 1, 1]
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> M.dot(v)
[6, 15, 24]
```

dual()

Returns the dual of a matrix, which is:

$(1/2) * levicivita(i, j, k, l) * M(k, l)$ 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.

eigenvals(***flags*)

Return eigen values using the berkowitz_eigenvals routine.

Since the roots routine doesn't always work well with Floats, they will be replaced with Rationals before calling that routine. If this is not desired, set flag `rational` to False.

eigenvects(***flags*)

Return list of triples (eigenval, multiplicity, basis).

The flag `simplify` has two effects: 1) if bool(simplify) is True, as_content_primitive() will be used to tidy up normalization artifacts; 2) if nullspace needs simplification to compute the basis, the simplify flag will be passed on to the nullspace routine which will interpret it there.

If the matrix contains any Floats, they will be changed to Rationals for computation purposes, but the answers will be returned after being evaluated with evalf. If it is desired to removed small imaginary portions during the evalf step, pass a value for the `chop` flag.

evalf(*prec=None*, ***options*)

Apply evalf() to each element of self.

exp()

Return the exponentiation of a square matrix.

expand(*deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)

Apply core.function.expand to each entry of the matrix.

Examples

```
>>> from sympy.abc import x
>>> from sympy.matrices import Matrix
>>> Matrix(1, 1, [x*(x+1)])
Matrix([[x*(x + 1)]])
>>> _.expand()
Matrix([[x**2 + x]])
```

extract(*rowsList*, *colsList*)

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range -n <= i < n where n is the number of rows or columns.

See Also:

`submatrix` (page 585)

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0,  1,  2],
```

```
[3,  4,  5],
[6,  7,  8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
[0,  1],
[3,  4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[8]])
```

get_diag_blocks()

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> A = Matrix([[1, 3, 0, 0], [y, z*z, 0, 0], [0, 0, x, 0], [0, 0, 0, 0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
Matrix([
[1,    3],
[y, z**2]])
>>> a2
Matrix([[x]])
>>> a3
Matrix([[0]])
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import Matrix, Float
>>> from sympy.abc import x, y
>>> A = Matrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
```

```
False
>>> A.has(Float)
True
```

classmethod hstack(**args*)

Return a matrix formed by joining args horizontally (i.e. by repeated application of row_join).

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2]])
```

integrate(**args*)

Integrate each element of the matrix.

See Also:

limit (page 578), diff (page 566)

Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.integrate((x, ))
Matrix([
[x**2/2, x*y],
[     x,   0]])
>>> M.integrate((x, 0, 2))
Matrix([
[2, 2*y],
[2,   0]])
```

inverse_ADJ(*iszerofunc=<function _iszero at 0x0000000004752E48>*)

Calculates the inverse using the adjugate matrix and a determinant.

See Also:

inv, inverse_LU (page 569), inverse_GE (page 569)

inverse_GE(*iszerofunc=<function _iszero at 0x0000000004752E48>*)

Calculates the inverse using Gaussian elimination.

See Also:

inv, inverse_LU (page 569), inverse_ADJ (page 569)

inverse_LU(*iszerofunc=<function _iszero at 0x0000000004752E48>*)

Calculates the inverse using LU decomposition.

See Also:

inv, inverse_GE (page 569), inverse_ADJ (page 569)

is_anti_symmetric(*simplify=True*)

Check if matrix M is an antisymmetric matrix, that is, M is a square matrix with all M[i, j] == -M[j, i].

When `simplify=True` (default), the sum M[i, j] + M[j, i] is simplified before testing to see if it is zero. By default, the SymPy simplify function is used. To use a custom function set simplify to a function that accepts a single argument which returns a simplified expression. To skip simplification, set simplify to False but note that although this will be faster, it may induce false negatives.

Examples

```
>>> from sympy import Matrix, symbols
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1],
[-1, 0]])
>>> m.is_anti_symmetric()
True
>>> x, y = symbols('x y')
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
>>> m
Matrix([
[ 0, 0, x],
[-y, 0, 0]])
>>> m.is_anti_symmetric()
False

>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [0, x**2 + 2*x + 1, y,
...                   -(x + 1)**2 , 0, x*y,
...                   -y, -x*y, 0])
```

Simplification of matrix elements is done by default so even though two elements which should be equal and opposite wouldn't pass an equality test, the matrix is still reported as anti-symmetric:

```
>>> m[0, 1] == -m[1, 0]
False
>>> m.is_anti_symmetric()
True
```

If 'simplify=False' is used for the case when a Matrix is already simplified, this will speed things up. Here, we see that without simplification the matrix does not appear anti-symmetric:

```
>>> m.is_anti_symmetric(simplify=False)
False
```

But if the matrix were already expanded, then it would appear anti-symmetric and simplification in the is_anti_symmetric routine is not needed:

```
>>> m = m.expand()
>>> m.is_anti_symmetric(simplify=False)
True
```

is_diagonal()

Check if matrix is diagonal, that is matrix in which the entries outside the main diagonal are all zero.

See Also:

is_lower (page 572), is_upper (page 575), is_diagonalizable (page 571), diagonalize (page 565)

Examples

```
>>> from sympy import Matrix, diag
>>> m = Matrix(2, 2, [1, 0, 0, 2])
>>> m
Matrix([
[1, 0],
[0, 2]])
>>> m.is_diagonal()
True

>>> m = Matrix(2, 2, [1, 1, 0, 2])
>>> m
Matrix([
[1, 1],
[0, 2]])
>>> m.is_diagonal()
False

>>> m = diag(1, 2, 3)
>>> m
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> m.is_diagonal()
True
```

is_diagonalizable(*reals_only=False*, *clear_subproducts=True*)

Check if matrix is diagonalizable.

If reals_only==True then check that diagonalized matrix consists of the only not complex values.

Some subproducts could be used further in other methods to avoid double calculations, By default (if clear_subproducts==True) they will be deleted.

See Also:

is_diagonal (page 570), diagonalize (page 565)

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
Matrix([
[1,  2, 0],
```

```
[0,  3, 0],
[2, -4, 2]])
>>> m.is_diagonalizable()
True
>>> m = Matrix(2, 2, [0, 1, 0, 0])
>>> m
Matrix([
[0, 1],
[0, 0]])
>>> m.is_diagonalizable()
False
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1],
[-1, 0]])
>>> m.is_diagonalizable()
True
>>> m.is_diagonalizable(True)
False
```

is_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.

See Also:

is_upper (page 575), is_diagonal (page 570), is_lower_hessenberg (page 573)

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_lower
True

>>> m = Matrix(4, 3, [0, 0, 0, 2, 0, 0, 1, 4 , 0, 6, 6, 5])
>>> m
Matrix([
[0, 0, 0],
[2, 0, 0],
[1, 4, 0],
[6, 6, 5]])
>>> m.is_lower
True

>>> from sympy.abc import x, y
>>> m = Matrix(2, 2, [x**2 + y, y**2 + x, 0, x + y])
>>> m
Matrix([
[x**2 + y, x + y**2],
[       0,    x + y]])
```

```
>>> m.is_lower
False
```

is_lower_hessenberg

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

See Also:

is_upper_hessenberg (page 575), is_lower (page 572)

Examples

```
>>> from sympy.matrices import Matrix
>>> a = Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1, 1]])
>>> a
Matrix([
[1, 2, 0, 0],
[5, 2, 3, 0],
[3, 4, 3, 7],
[5, 6, 1, 1]])
>>> a.is_lower_hessenberg
True
```

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
```

is_square

Checks if a matrix is square.

A matrix is square if the number of rows equals the number of columns. The empty matrix is square by definition, since the number of rows and the number of columns are both zero.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 3], [4, 5, 6]])
>>> b = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> c = Matrix([])
>>> a.is_square
```

```
False
>>> b.is_square
True
>>> c.is_square
True
```

is_symbolic()

Checks if any elements contain Symbols.

Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.is_symbolic()
True
```

is_symmetric(*simplify=True*)

Check if matrix is symmetric matrix, that is square matrix and is equal to its transpose.

By default, simplifications occur before testing symmetry. They can be skipped using 'simplify=False'; while speeding things a bit, this may however induce false negatives.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [0, 1, 1, 2])
>>> m
Matrix([
[0, 1],
[1, 2]])
>>> m.is_symmetric()
True

>>> m = Matrix(2, 2, [0, 1, 2, 0])
>>> m
Matrix([
[0, 1],
[2, 0]])
>>> m.is_symmetric()
False

>>> m = Matrix(2, 3, [0, 0, 0, 0, 0, 0])
>>> m
Matrix([
[0, 0, 0],
[0, 0, 0]])
>>> m.is_symmetric()
False

>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [1, x**2 + 2*x + 1, y, (x + 1)**2 , 2, 0, y, 0, 3])
>>> m
```

```
Matrix([
[         1, x**2 + 2*x + 1, y],
[(x + 1)**2,              2, 0],
[         y,              0, 3]])
>>> m.is_symmetric()
True
```

If the matrix is already simplified, you may speed-up is_symmetric() test by using 'simplify=False'.

```
>>> m.is_symmetric(simplify=False)
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True
```

is_upper

Check if matrix is an upper triangular matrix. True can be returned even if the matrix is not square.

See Also:

is_lower (page 572), is_diagonal (page 570), is_upper_hessenberg (page 575)

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_upper
True

>>> m = Matrix(4, 3, [5, 1, 9, 0, 4 , 6, 0, 0, 5, 0, 0, 0])
>>> m
Matrix([
[5, 1, 9],
[0, 4, 6],
[0, 0, 5],
[0, 0, 0]])
>>> m.is_upper
True

>>> m = Matrix(2, 3, [4, 2, 5, 6, 1, 1])
>>> m
Matrix([
[4, 2, 5],
[6, 1, 1]])
>>> m.is_upper
False
```

is_upper_hessenberg

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

See Also:

is_lower_hessenberg (page 573), is_upper (page 575)

Examples

```
>>> from sympy.matrices import Matrix
>>> a = Matrix([[1, 4, 2, 3], [3, 4, 1, 7], [0, 2, 3, 4], [0, 0, 1, 3]])
>>> a
Matrix([
[1, 4, 2, 3],
[3, 4, 1, 7],
[0, 2, 3, 4],
[0, 0, 1, 3]])
>>> a.is_upper_hessenberg
True
```

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.

Examples

```
>>> from sympy import Matrix, zeros
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
```

jacobian(X)

Calculates the Jacobian matrix (derivative of a vectorial function).

Parameters **self** : vector of expressions representing functions f_i(x_1, ..., x_n).

X : set of x_i's in order, it can be a list or a Matrix

Both self and X can be a row or a column matrix in any order :

(i.e., jacobian() should always work). :

See Also:

hessian, wronskian

Examples

```
>>> from sympy import sin, cos, Matrix
>>> from sympy.abc import rho, phi
>>> X = Matrix([rho*cos(phi), rho*sin(phi), rho**2])
>>> Y = Matrix([rho, phi])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi),  rho*cos(phi)],
[   2*rho,             0]])
>>> X = Matrix([rho*cos(phi), rho*sin(phi)])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi),  rho*cos(phi)]])
```

jordan_cells(*calc_transformation=True*)

Return a list of Jordan cells of current matrix. This list shape Jordan matrix J.

If calc_transformation is specified as False, then transformation P such that

J = P^-1 * M * P

will not be calculated.

See Also:

jordan_form (page 577)

Notes

Calculation of transformation P is not implemented yet.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 4, [
...  6,  5, -2, -3,
... -3, -1,  3,  3,
...  2,  1, -2, -3,
... -1,  1,  5,  5])

>>> (P, Jcells) = m.jordan_cells()
>>> Jcells[0]
Matrix([
[2, 1],
[0, 2]])
>>> Jcells[1]
Matrix([
[2, 1],
[0, 2]])
```

jordan_form(*calc_transformation=True*)

Return Jordan form J of current matrix.

If calc_transformation is specified as False, then transformation P such that

J = P^-1 * M * P

will not be calculated.

See Also:

jordan_cells (page 577)

Notes

Calculation of transformation P is not implemented yet.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix([
...        [ 6,  5, -2, -3],
...        [-3, -1,  3,  3],
...        [ 2,  1, -2, -3],
...        [-1,  1,  5,  5]])
>>> (P, J) = m.jordan_form()
>>> J
Matrix([
[2, 1, 0, 0],
[0, 2, 0, 0],
[0, 0, 2, 1],
[0, 0, 0, 2]])
```

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 578)

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 578)

limit(**args*)

Calculate the limit of each element in the matrix.

See Also:

integrate (page 569), diff (page 566)

Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

lower_triangular_solve(*rhs*)
Solves Ax = B, where A is a lower triangular matrix.

See Also:

upper_triangular_solve (page 586), cholesky_solve (page 563), diagonal_solve (page 565), LDLsolve (page 557), LUsolve (page 558), QRsolve (page 559)

minorEntry(*i*, *j*, *method='berkowitz'*)
Calculate the minor of an element.

See Also:

minorMatrix (page 579), cofactor (page 563), cofactorMatrix (page 563)

minorMatrix(*i*, *j*)
Creates the minor matrix of a given element.

See Also:

minorEntry (page 579), cofactor (page 563), cofactorMatrix (page 563)

multiply(*b*)
Returns self*b

See Also:

dot (page 566), cross (page 564), multiply_elementwise (page 579)

multiply_elementwise(*b*)
Return the Hadamard product (elementwise product) of A and B

See Also:

cross (page 564), dot (page 566), multiply (page 579)

Examples

```
>>> from sympy.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> A.multiply_elementwise(B)
Matrix([
[  0, 10, 200],
[300, 40,   5]])
```

n(*prec=None*, ***options*)
Apply evalf() to each element of self.

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	–	max(abs(x))
-inf	–	min(abs(x))
1	–	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 580)

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(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
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1
```

normalized()

Return the normalized version of `self`.

See Also:

norm (page 579)

nullspace(*simplified=False, simplify=False*)

Returns list of vectors (Matrix objects) that span nullspace of self

permuteBkwd(*perm*)

Permute the rows of the matrix with the given permutation in reverse.

See Also:

permuteFwd (page 581)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.permuteBkwd([[0, 1], [0, 2]])
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

permuteFwd(*perm*)

Permute the rows of the matrix with the given permutation.

See Also:

permuteBkwd (page 580)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.permuteFwd([[0, 1], [0, 2]])
Matrix([
[0, 0, 1],
[1, 0, 0],
[0, 1, 0]])
```

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]])
```

rank(*simplified=False, iszerofunc=<function _iszero at 0x0000000004752E48>, simplify=False*)

Returns the rank of a matrix

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2
```

replace(*F, G, map=False*)

Replaces Function F in Matrix entries with Function G.

Examples

```
>>> from sympy import symbols, Function, Matrix
>>> F, G = symbols('F, G', cls=Function)
>>> M = Matrix(2, 2, lambda i, j: F(i+j)) ; M
Matrix([
[F(0), F(1)],
[F(1), F(2)]])
>>> N = M.replace(F,G)
>>> N
Matrix([
[G(0), G(1)],
[G(1), G(2)]])
```

row_insert(*pos, mti*)

Insert one or more rows at the given row position.

See Also:

row, col_insert (page 563)

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([
[0, 0, 0],
```

```
[1, 1, 1],
[0, 0, 0],
[0, 0, 0]])
```

row_join(*rhs*)

Concatenates two matrices along self's last and rhs's first column

See Also:

row, col_join (page 563)

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.row_join(V)
Matrix([
[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

rref(*simplified=False, iszerofunc=<function _iszero at 0x0000000004752E48>, simplify=False*)

Return reduced row-echelon form of matrix and indices of pivot vars.

To simplify elements before finding nonzero pivots set simplify=True (to use the default SymPy simplify function) or pass a custom simplify function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([
[1, 0],
[0, 1]]), [0, 1])
```

shape

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

Examples

```
>>> from sympy.matrices import zeros
>>> M = zeros(2, 3)
>>> M.shape
(2, 3)
>>> M.rows
2
>>> M.cols
3
```

simplify(*ratio=1.7, measure=<function count_ops at 0x0000000004035128>*)

Apply simplify to each element of the matrix.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import sin, cos
>>> from sympy.matrices import SparseMatrix
>>> SparseMatrix(1, 1, [x*sin(y)**2 + x*cos(y)**2])
Matrix([[x*sin(y)**2 + x*cos(y)**2]])
>>> _.simplify()
Matrix([[x]])
```

singular_values()

Compute the singular values of a Matrix

See Also:

condition_number (page 564)

Examples

```
>>> from sympy import Matrix, Symbol
>>> x = Symbol('x', real=True)
>>> A = Matrix([[0, 1, 0], [0, x, 0], [-1, 0, 0]])
>>> A.singular_values()
[sqrt(x**2 + 1), 1, 0]
```

solve(*rhs, method='GE'*)

Return solution to self*soln = 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
```

submatrix(*keys*)

Get a slice/submatrix of the matrix using the given slice.

See Also:

extract (page 567)

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 4, lambda i, j: i+j)
>>> m
Matrix([
[0, 1, 2, 3],
[1, 2, 3, 4],
[2, 3, 4, 5],
[3, 4, 5, 6]])
>>> m[:1, 1]
Matrix([[1]])
>>> m[:2, :1]
Matrix([
[0],
[1]])
>>> m[2:4, 2:4]
Matrix([
[4, 5],
[5, 6]])
```

subs(**args, **kwargs*)

Return a new matrix with subs applied to each entry.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.matrices import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.subs(x, y)
Matrix([[y]])
>>> Matrix(_).subs(y, x)
Matrix([[x]])
```

table(*printer, 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())

`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]
```

upper_triangular_solve(*rhs*)

Solves Ax = B, where A is an upper triangular matrix.

See Also:

`lower_triangular_solve` (page 578), `cholesky_solve` (page 563), `diagonal_solve` (page 565), `LDLsolve` (page 557), `LUsolve` (page 558), `QRsolve` (page 559)

values()

Return non-zero values of self.

vec()

Return the Matrix converted into a one column matrix by stacking columns

See Also:

vech (page 587)

Examples

```
>>> from sympy import Matrix
>>> m=Matrix([[1, 3], [2, 4]])
>>> m
Matrix([
[1, 3],
[2, 4]])
>>> m.vec()
Matrix([
[1],
[2],
[3],
[4]])
```

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 (page 587)

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]])
```

classmethod vstack(**args*)

Return a matrix formed by joining args vertically (i.e. by repeated application of col_join).

Examples

```
>>> from sympy.matrices import Matrix, eye
>>> Matrix.vstack(eye(2), 2*eye(2))
Matrix([
[1, 0],
[0, 1],
[2, 0],
[0, 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.ImmutableMatrix'>
```

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**(*r, c=None, cls=None*)

Returns a matrix of zeros with `r` rows and `c` columns; if `c` is omitted a square matrix will be returned.

See Also:

ones, eye, diag

sympy.matrices.dense.**ones**(*r*, *c=None*)

Returns a matrix of ones with r rows and c columns; if c is omitted a square matrix will be returned.

See Also:

zeros, eye, diag

sympy.matrices.dense.**eye**(*n*, *cls=None*)

Create square identity matrix n x n

See Also:

diag, zeros, ones

sympy.matrices.dense.**diag**(**values*, ***kwargs*)

Create a sparse, diagonal matrix from a list of diagonal values.

See Also:

eye

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([
[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.ImmutableMatrix'>
```

`sympy.matrices.dense.`**`jordan_cell`**(*eigenval, n*)

Create matrix of Jordan cell kind:

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

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]))
[                   d               d            ]
[     0        0    --(g(x, y))     --(g(x, y))  ]
[                   dx              dy           ]
[                                                ]
[     0        0        2*x              3       ]
[                                                ]
[                     2               2          ]
[d                   d               d           ]
[--(g(x, y))  2*x   ---(f(x, y))   -----(f(x, y))]
[dx                  2            dy dx         ]
[                   dx                           ]
[                                                ]
[                     2               2          ]
[d                   d               d           ]
[--(g(x, y))   3   -----(f(x, y))   ---(f(x, y)) ]
[dy                dy dx               2         ]
[                                     dy         ]
```

sympy.matrices.dense.**GramSchmidt**(*vlist*, *orthog=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='bareis'*)

Compute Wronskian for [] of functions

```
                   | f1       f2        ...   fn      |
                   | f1'      f2'       ...   fn'     |
                   |  .        .        .      .      |
W(f1, ..., fn) = |  .        .         .     .      |
                   |  .        .          .    .      |
                   |  (n)      (n)            (n)     |
                   | D   (f1) D   (f2)  ...  D   (fn) |
```

see: http://en.wikipedia.org/wiki/Wronskian

See Also:

sympy.matrices.mutable.Matrix.jacobian, hessian

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, ..., z), do not vanish for n = 0.

Casoratian is defined by k x k determinant:

```
+  a(n)     b(n)     . . . z(n)     +
|  a(n+1)   b(n+1)   . . . z(n+1)   |
|    .         .     .        .     |
|    .         .       .      .     |
|    .         .         .    .     |
+  a(n+k-1) b(n+k-1) . . . z(n+k-1) +
```

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*)

Create random matrix with dimensions `r` x `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.

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)
[0, 68, 43]
[0, 68,  0]
[0, 91, 34]
```

Numpy Utility Functions Reference

`sympy.matrices.dense.`**`list2numpy`**(*l*)

Converts python list of SymPy expressions to a NumPy array.

See Also:

`matrix2numpy`

`sympy.matrices.dense.`**`matrix2numpy`**(*m*)

Converts SymPy's matrix to a NumPy array.

See Also:

`list2numpy`

`sympy.matrices.dense.`**`symarray`**(*prefix*, *shape*)

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.

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] is b[0]
True
>>> a = symarray('a', 3)
>>> b = symarray('b', 3)
>>> a[0] is 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]]]
```

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 Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

rot_axis3 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 theta (in radians) about the 2-axis.

See Also:

rot_axis1 Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

rot_axis3 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_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 theta (in radians) about the 3-axis.

See Also:

rot_axis1 Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

rot_axis2 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.

5.14.2 Dense Matrices

Matrix Class Reference

class sympy.matrices.dense.**MutableDenseMatrix**

col_del(*i*)

Delete the given column.

See Also:

col, row_del

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

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

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

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

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`, `ones`

row_del(*i*)

Delete the given row.

See Also:

`row`, `col_del`

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`, `col_op`

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`

Examples

```
>>> from sympy.matrices import Matrix
>>> M = Matrix([[0, 1], [1, 0]])
>>> M
Matrix([
[0, 1],
[1, 0]])
>>> M.row_swap(0, 1)
>>> M
Matrix([
[1, 0],
[0, 1]])
```

simplify(*ratio=1.7, measure=<function count_ops at 0x0000000004035128>*)

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 1182)

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`, `col_op`

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.**ImmutableMatrix**

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
```

ImmutableMatrix.**C**

By-element conjugation.

ImmutableMatrix.**adjoint**()

Conjugate transpose or Hermitian conjugation.

ImmutableMatrix.**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]])
```

ImmutableMatrix.**equals**(*other*, *failing_expression=False*)

Applies `equals` to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if failing_expression is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See Also:

`sympy.core.expr.equals`

Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x
>>> from sympy import cos
>>> A = Matrix([x*(x - 1), 0])
>>> B = Matrix([x**2 - x, 0])
>>> A == B
False
>>> A.simplify() == B.simplify()
True
>>> A.equals(B)
True
>>> A.equals(2)
False
```

5.14.3 Sparse Matrices

SparseMatrix Class Reference

class sympy.matrices.sparse.**SparseMatrix**(**args*)

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]])
```

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

add(*other*)

Add two sparse matrices with dictionary representation.

See Also:

multiply (page 604)

Examples

```
>>> from sympy.matrices import SparseMatrix, eye, ones
>>> SparseMatrix(eye(3)).add(SparseMatrix(ones(3)))
Matrix([
[2, 1, 1],
[1, 2, 1],
[1, 1, 2]])
>>> SparseMatrix(eye(3)).add(-SparseMatrix(eye(3)))
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
```

Only the non-zero elements are stored, so the resulting dictionary that is used to represent the sparse matrix is empty: >>> _._smat {}

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(*j*)

Returns column j from self as a column vector.

See Also:

row (page 605), col_list (page 603)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.col(0)
Matrix([
[1],
[3]])
```

col_list()

Returns a column-sorted list of non-zero elements of the matrix.

See Also:

col_op, row_list (page 605)

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)]
```

classmethod eye(*n*)

Return an n x n identity matrix.

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import SparseMatrix, Float
>>> from sympy.abc import x, y
>>> A = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
```

is_symmetric(*simplify=True*)

Return True if self is symmetric.

Examples

```
>>> from sympy.matrices import SparseMatrix, eye
>>> M = SparseMatrix(eye(3))
>>> M.is_symmetric()
True
>>> M[0, 2] = 1
>>> M.is_symmetric()
False
```

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 Grondelle (1999) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582, downloaded from http://tinyurl.com/9o2jsxj

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])
```

multiply(*other*)

Fast multiplication exploiting the sparsity of the matrix.

See Also:

add (page 601)

Examples

```
>>> from sympy.matrices import SparseMatrix, ones
>>> A, B = SparseMatrix(ones(4, 3)), SparseMatrix(ones(3, 4))
>>> A.multiply(B) == 3*ones(4)
True
```

nnz()

Returns the number of non-zero elements in Matrix.

reshape(*rows, cols*)

Reshape matrix while retaining original size.

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix(4, 2, range(8))
>>> S.reshape(2, 4)
Matrix([
[0, 1, 2, 3],
[4, 5, 6, 7]])
```

row(*i*)

Returns column i from self as a row vector.

See Also:

col (page 603), row_list (page 605)

Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.row(0)
Matrix([[1, 2]])
```

row_list()

Returns a row-sorted list of non-zero elements of the matrix.

See Also:

row_op, col_list (page 603)

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 factororization.

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Grondelle (1999) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582, downloaded from http://tinyurl.com/9o2jsxj

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.row_structure_symbolic_cholesky()
[[0], [], [0], [1, 2]]
```

scalar_multiply(*scalar*)

Scalar element-wise multiplication

solve(*rhs, method='LDL'*)

Return solution to self*soln = 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
```

tolist()

Convert this sparse matrix into a list of nested Python lists.

Examples

```
>>> from sympy.matrices import SparseMatrix, ones
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.tolist()
[[1, 2], [3, 4]]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> SparseMatrix(ones(0, 3)).tolist()
[]
```

classmethod zeros(*r, c=None*)

Return an r x c matrix of zeros, square if c is omitted.

class sympy.matrices.sparse.**MutableSparseMatrix**(**args*)

col_del(*k*)

Delete the given column of the matrix.

See Also:

row_del (page 609)

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([
[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 (self[i, j], i) for i in range(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 607)

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]])
>>> 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 611), `col_op` (page 608)

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 610), `col_op` (page 608)

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**(**args*)

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)
```

subs(**args, **kwargs*)

Return a new matrix with subs applied to each entry.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.matrices import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.subs(x, y)
Matrix([[y]])
>>> Matrix(_).subs(y, x)
Matrix([[x]])
```

5.14.4 Immutable Matrices

The standard `Matrix` class in SymPy is mutable. This is important for performance reasons but means that standard matrices can not 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 `ImmutableMatrix` class 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.**ImmutableMatrix**

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
```

C

By-element conjugation.

adjoint()

Conjugate transpose or Hermitian conjugation.

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]])
```

equals(*other*, *failing_expression=False*)

Applies `equals` to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if failing_expression is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See Also:

`sympy.core.expr.equals`

Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x
>>> from sympy import cos
>>> A = Matrix([x*(x - 1), 0])
>>> B = Matrix([x**2 - x, 0])
>>> A == B
False
>>> A.simplify() == B.simplify()
True
>>> A.equals(B)
```

```
True
>>> A.equals(2)
False
```

5.14.5 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'^-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 615)'s rather than scalar symbols.

Matrix Expressions Core Reference

class sympy.matrices.expressions.**MatrixExpr**

Matrix Expression Class Matrix Expressions subclass SymPy Expr's so that MatAdd inherits from Add MatMul inherits from Mul MatPow inherits from Pow

They use _op_priority to gain control with binary operations (+, , -, *) are used

They implement operations specific to Matrix Algebra.

Attributes

is_Identity	

T

Matrix transposition.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type ImmutableMatrix.

See Also:

`as_mutable` **(page 615)** 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 614)** returns ImmutableMatrix

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
```

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)
2*A*B + I
```

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'
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B)'
>>> transpose(A*B)
B'*A'
```

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 614) 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 + Z*Y]])
```

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',  0],
[Z', Y']])
>>> _.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, Ma
>>> 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 + Z*Y]])
```

5.15 Welcome to mpmath's documentation!

Mpmath is a Python library for arbitrary-precision floating-point arithmetic. For general information about mpmath, see the project website http://code.google.com/p/mpmath/

These documentation pages include general information as well as docstring listing with extensive use of examples that can be run in the interactive Python interpreter. For quick access to the docstrings of individual functions, use the *genindex*, or type `help(mpmath.function_name)` in the Python interactive prompt.

5.15.1 Introduction

Setting up mpmath

Download and installation

Installer The mpmath setup files can be downloaded from the mpmath download page or the Python Package Index. Download the source package (available as both .zip and .tar.gz), extract it, open the extracted directory, and run

```
python setup.py install
```

If you are using Windows, you can download the binary installer

```
mpmath-(version).win32.exe
```

from the mpmath website or the Python Package Index. Run the installer and follow the instructions.

Using setuptools If you have setuptools installed, you can download and install mpmath in one step by running:

```
easy_install mpmath
```

or

```
python -m easy_install mpmath
```

If you have an old version of mpmath installed already, you may have to pass `easy_install` the `-U` flag to force an upgrade.

Debian/Ubuntu Debian and Ubuntu users can install mpmath with

```
sudo apt-get install python-mpmath
```

See debian and ubuntu package information; please verify that you are getting the latest version.

OpenSUSE Mpmath is provided in the "Science" repository for all recent versions of openSUSE. To add this repository to the YAST software management tool, see http://en.opensuse.org/SDB:Add_package_repositories

Look up http://download.opensuse.org/repositories/science/ for a list of supported OpenSUSE versions and use http://download.opensuse.org/repositories/science/openSUSE_12.2/ (or accordingly for your OpenSUSE version) as the repository URL for YAST.

Current development version See http://code.google.com/p/mpmath/source/checkout for instructions on how to check out the mpmath Subversion repository. The source code can also be browsed online from the Google Code page.

Checking that it works After the setup has completed, you should be able to fire up the interactive Python interpreter and do the following:

```
>>> from mpmath import *
>>> mp.dps = 50
>>> print mpf(2) ** mpf('0.5')
1.4142135623730950488016887242096980785696718753769
>>> print 2*pi
6.2831853071795864769252867665590057683943387987502
```

Note: if you have are upgrading mpmath from an earlier version, you may have to manually uninstall the old version or remove the old files.

Using gmpy (optional)

By default, mpmath uses Python integers internally. If gmpy version 1.03 or later is installed on your system, mpmath will automatically detect it and transparently use gmpy integers intead. This makes mpmath much faster, especially at high precision (approximately above 100 digits).

To verify that mpmath uses gmpy, check the internal variable `BACKEND` is not equal to 'python':

```
>>> import mpmath.libmp
>>> mpmath.libmp.BACKEND
'gmpy'
```

The gmpy mode can be disabled by setting the MPMATH_NOGMPY environment variable. Note that the mode cannot be switched during runtime; mpmath must be re-imported for this change to take effect.

Running tests

It is recommended that you run mpmath's full set of unit tests to make sure everything works. The tests are located in the `tests` subdirectory of the main mpmath directory. They can be run in the interactive interpreter using the `runtests()` function:

```
import mpmath
mpmath.runtests()
```

Alternatively, they can be run from the `tests` directory via

```
python runtests.py
```

The tests should finish in about a minute. If you have psyco installed, the tests can also be run with

```
python runtests.py -psyco
```

which will cut the running time in half.

If any test fails, please send a detailed bug report to the mpmath issue tracker. The tests can also be run with py.test. This will sometimes generate more useful information in case of a failure.

To run the tests with support for gmpy disabled, use

```
python runtests.py -nogmpy
```

To enable extra diagnostics, use

```
python runtests.py -strict
```

Compiling the documentation

If you downloaded the source package, the text source for these documentation pages is included in the `doc` directory. The documentation can be compiled to pretty HTML using Sphinx. Go to the `doc` directory and run

```
python build.py
```

You can also test that all the interactive examples in the documentation work by running

```
python run_doctest.py
```

and by running the individual `.py` files in the mpmath source.

(The doctests may take several minutes.)

Finally, some additional demo scripts are available in the `demo` directory included in the source package.

Mpmath under SymPy

Mpmath is available as a subpackage of SymPy. With SymPy installed, you can just do

```
import sympy.mpmath as mpmath
```

instead of `import mpmath`. Note that the SymPy version of mpmath might not be the most recent. You can make a separate mpmath installation even if SymPy is installed; the two mpmath packages will not interfere with each other.

Mpmath under Sage

Mpmath is a standard package in Sage, in version 4.1 or later of Sage. Mpmath is preinstalled a regular Python module, and can be imported as usual within Sage:

```
----------------------------------------------------------------------
| Sage Version 4.1, Release Date: 2009-07-09                         |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
sage: import mpmath
sage: mpmath.mp.dps = 50
sage: print mpmath.mpf(2) ** 0.5
1.4142135623730950488016887242096980785696718753769
```

The mpmath installation under Sage automatically use Sage integers for asymptotically fast arithmetic, so there is no need to install GMPY:

```
sage: mpmath.libmp.BACKEND
'sage'
```

In Sage, mpmath can alternatively be imported via the interface library `sage.libs.mpmath.all`. For example:

```
sage: import sage.libs.mpmath.all as mpmath
```

This module provides a few extra conversion functions, including `call()` which permits calling any mpmath function with Sage numbers as input, and getting Sage `RealNumber` or `ComplexNumber` instances with the appropriate precision back:

```
sage: w = mpmath.call(mpmath.erf, 2+3*I, prec=100)
sage: w
-20.829461427614568389103088452 + 8.6873182714701631444280787545*I
sage: type(w)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: w.prec()
100
```

See the help for `sage.libs.mpmath.all` for further information.

Basic usage

In interactive code examples that follow, it will be assumed that all items in the `mpmath` namespace have been imported:

```
>>> from mpmath import *
```

Importing everything can be convenient, especially when using mpmath interactively, but be careful when mixing mpmath with other libraries! To avoid inadvertently overriding other functions or objects, explicitly import only the needed objects, or use the `mpmath.` or `mp.` namespaces:

```
from mpmath import sin, cos
sin(1), cos(1)

import mpmath
mpmath.sin(1), mpmath.cos(1)
```

```
from mpmath import mp    # mp context object -- to be explained
mp.sin(1), mp.cos(1)
```

Number types

Mpmath provides the following numerical types:

Class	Description
`mpf`	Real float
`mpc`	Complex float
`matrix`	Matrix

The following section will provide a very short introduction to the types `mpf` and `mpc`. Intervals and matrices are described further in the documentation chapters on interval arithmetic and matrices / linear algebra.

The `mpf` type is analogous to Python's built-in `float`. It holds a real number or one of the special values `inf` (positive infinity), `-inf` (negative infinity) and `nan` (not-a-number, indicating an indeterminate result). You can create `mpf` instances from strings, integers, floats, and other `mpf` instances:

```
>>> mpf(4)
mpf('4.0')
>>> mpf(2.5)
mpf('2.5')
>>> mpf("1.25e6")
mpf('1250000.0')
>>> mpf(mpf(2))
mpf('2.0')
>>> mpf("inf")
mpf('+inf')
```

The `mpc` type represents a complex number in rectangular form as a pair of `mpf` instances. It can be constructed from a Python `complex`, a real number, or a pair of real numbers:

```
>>> mpc(2,3)
mpc(real='2.0', imag='3.0')
>>> mpc(complex(2,3)).imag
mpf('3.0')
```

You can mix `mpf` and `mpc` instances with each other and with Python numbers:

```
>>> mpf(3) + 2*mpf('2.5') + 1.0
mpf('9.0')
>>> mp.dps = 15      # Set precision (see below)
>>> mpc(1j)**0.5
mpc(real='0.70710678118654757', imag='0.70710678118654757')
```

Setting the precision

Mpmath uses a global working precision; it does not keep track of the precision or accuracy of individual numbers. Performing an arithmetic operation or calling `mpf()` rounds the result to the current working precision. The working precision is controlled by a context object called `mp`, which has the following default state:

```
>>> print mp
Mpmath settings:
  mp.prec = 53                [default: 53]
  mp.dps = 15                 [default: 15]
  mp.trap_complex = False     [default: False]
```

The term **prec** denotes the binary precision (measured in bits) while **dps** (short for *decimal places*) is the decimal precision. Binary and decimal precision are related roughly according to the formula `prec = 3.33*dps`. For example, it takes a precision of roughly 333 bits to hold an approximation of pi that is accurate to 100 decimal places (actually slightly more than 333 bits is used).

Changing either precision property of the `mp` object automatically updates the other; usually you just want to change the `dps` value:

```
>>> mp.dps = 100
>>> mp.dps
100
>>> mp.prec
336
```

When the precision has been set, all `mpf` operations are carried out at that precision:

```
>>> mp.dps = 50
>>> mpf(1) / 6
mpf('0.16666666666666666666666666666666666666666666666666656')
>>> mp.dps = 25
>>> mpf(2) ** mpf('0.5')
mpf('1.4142135623730950488016887 13')
```

The precision of complex arithmetic is also controlled by the `mp` object:

```
>>> mp.dps = 10
>>> mpc(1,2) / 3
mpc(real='0.3333333333321', imag='0.6666666666642')
```

There is no restriction on the magnitude of numbers. An `mpf` can for example hold an approximation of a large Mersenne prime:

```
>>> mp.dps = 15
>>> print mpf(2)**32582657 - 1
1.24575026015369e+9808357
```

Or why not 1 googolplex:

```
>>> print mpf(10) ** (10**100)
1.0e+100000000000000000000000000000000000000000000000000...
```

The (binary) exponent is stored exactly and is independent of the precision.

Temporarily changing the precision It is often useful to change the precision during only part of a calculation. A way to temporarily increase the precision and then restore it is as follows:

```
>>> mp.prec += 2
>>> # do_something()
>>> mp.prec -= 2
```

In Python 2.5, the `with` statement along with the mpmath functions `workprec`, `workdps`, `extraprec` and `extradps` can be used to temporarily change precision in a more safe manner:

```
>>> from __future__ import with_statement
>>> with workdps(20):
...     print mpf(1)/7
...     with extradps(10):
...         print mpf(1)/7
...
0.14285714285714285714
0.14285714285714285714285714285714285714
>>> mp.dps
15
```

The `with` statement ensures that the precision gets reset when exiting the block, even in the case that an exception is raised. (The effect of the `with` statement can be emulated in Python 2.4 by using a `try/finally` block.)

The `workprec` family of functions can also be used as function decorators:

```
>>> @workdps(6)
... def f():
...     return mpf(1)/3
...
>>> f()
mpf('0.33333331346511841')
```

Some functions accept the `prec` and `dps` keyword arguments and this will override the global working precision. Note that this will not affect the precision at which the result is printed, so to get all digits, you must either use increase precision afterward when printing or use `nstr/nprint`:

```
>>> mp.dps = 15
>>> print exp(1)
2.71828182845905
>>> print exp(1, dps=50)      # Extra digits won't be printed
2.71828182845905
>>> nprint(exp(1, dps=50), 50)
2.7182818284590452353602874713526624977572470937
```

Finally, instead of using the global context object `mp`, you can create custom contexts and work with methods of those instances instead of global functions. The working precision will be local to each context object:

```
>>> mp2 = mp.clone()
>>> mp.dps = 10
>>> mp2.dps = 20
>>> print mp.mpf(1) / 3
0.3333333333
>>> print mp2.mpf(1) / 3
0.33333333333333333333
```

Note: the ability to create multiple contexts is a new feature that is only partially implemented. Not all mpmath functions are yet available as context-local methods. In the present version, you are likely to encounter bugs if you try mixing different contexts.

Providing correct input

Note that when creating a new `mpf`, the value will at most be as accurate as the input. *Be careful when mixing mpmath numbers with Python floats*. When working at high precision, fractional `mpf` values should be created from strings or integers:

```
>>> mp.dps = 30
>>> mpf(10.9)   # bad
mpf('10.900000000000000355271367880050 1')
>>> mpf('10.9')  # good
mpf('10.899999999999999999999999999997')
>>> mpf(109) / mpf(10)   # also good
mpf('10.899999999999999999999999999997')
>>> mp.dps = 15
```

(Binary fractions such as 0.5, 1.5, 0.75, 0.125, etc, are generally safe as input, however, since those can be represented exactly by Python floats.)

Printing

By default, the `repr()` of a number includes its type signature. This way `eval` can be used to recreate a number from its string representation:

```
>>> eval(repr(mpf(2.5)))
mpf('2.5')
```

Prettier output can be obtained by using `str()` or `print`, which hide the `mpf` and `mpc` signatures and also suppress rounding artifacts in the last few digits:

```
>>> mpf("3.14159")
mpf('3.1415899999999999')
>>> print mpf("3.14159")
3.14159
>>> print mpc(1j)**0.5
(0.707106781186548 + 0.707106781186548j)
```

Setting the `mp.pretty` option will use the `str()`-style output for `repr()` as well:

```
>>> mp.pretty = True
>>> mpf(0.6)
0.6
>>> mp.pretty = False
>>> mpf(0.6)
mpf('0.59999999999999998')
```

The number of digits with which numbers are printed by default is determined by the working precision. To specify the number of digits to show without changing the working precision, use `mpmath.nstr()` (page 634) and `mpmath.nprint()` (page 634):

```
>>> a = mpf(1) / 6
>>> a
mpf('0.16666666666666666')
>>> nstr(a, 8)
'0.16666667'
>>> nprint(a, 8)
0.16666667
```

```
>>> nstr(a, 50)
'0.16666666666666665741480812812369549646973609924316'
```

5.15.2 Basic features

Contexts

High-level code in mpmath is implemented as methods on a "context object". The context implements arithmetic, type conversions and other fundamental operations. The context also holds settings such as precision, and stores cache data. A few different contexts (with a mostly compatible interface) are provided so that the high-level algorithms can be used with different implementations of the underlying arithmetic, allowing different features and speed-accuracy tradeoffs. Currently, mpmath provides the following contexts:

- Arbitrary-precision arithmetic (`mp`)
- A faster Cython-based version of `mp` (used by default in Sage, and currently only available there)
- Arbitrary-precision interval arithmetic (`iv`)
- Double-precision arithmetic using Python's builtin `float` and `complex` types (`fp`)

Most global functions in the global mpmath namespace are actually methods of the `mp` context. This fact is usually transparent to the user, but sometimes shows up in the form of an initial parameter called "ctx" visible in the help for the function:

```
>>> import mpmath
>>> help(mpmath.fsum)
Help on method fsum in module mpmath.ctx_mp_python:

fsum(ctx, terms, absolute=False, squared=False) method of mpmath.ctx_mp.MPContext instance
    Calculates a sum containing a finite number of terms (for infinite
    series, see :func:`~mpmath.nsum`). The terms will be converted to
...
```

The following operations are equivalent:

```
>>> mpmath.mp.dps = 15; mpmath.mp.pretty = False
>>> mpmath.fsum([1,2,3])
mpf('6.0')
>>> mpmath.mp.fsum([1,2,3])
mpf('6.0')
```

The corresponding operation using the `fp` context:

```
>>> mpmath.fp.fsum([1,2,3])
6.0
```

Common interface

`ctx.mpf` creates a real number:

```
>>> from mpmath import mp, fp
>>> mp.mpf(3)
mpf('3.0')
```

```
>>> fp.mpf(3)
3.0
```

`ctx.mpc` creates a complex number:

```
>>> mp.mpc(2,3)
mpc(real='2.0', imag='3.0')
>>> fp.mpc(2,3)
(2+3j)
```

`ctx.matrix` creates a matrix:

```
>>> mp.matrix([[1,0],[0,1]])
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
>>> _[0,0]
mpf('1.0')
>>> fp.matrix([[1,0],[0,1]])
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
>>> _[0,0]
1.0
```

`ctx.prec` holds the current precision (in bits):

```
>>> mp.prec
53
>>> fp.prec
53
```

`ctx.dps` holds the current precision (in digits):

```
>>> mp.dps
15
>>> fp.dps
15
```

`ctx.pretty` controls whether objects should be pretty-printed automatically by `repr()`. Pretty-printing for `mp` numbers is disabled by default so that they can clearly be distinguished from Python numbers and so that `eval(repr(x)) == x` works:

```
>>> mp.mpf(3)
mpf('3.0')
>>> mpf = mp.mpf
>>> eval(repr(mp.mpf(3)))
mpf('3.0')
>>> mp.pretty = True
>>> mp.mpf(3)
3.0
>>> fp.matrix([[1,0],[0,1]])
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
>>> fp.pretty = True
>>> fp.matrix([[1,0],[0,1]])
[1.0  0.0]
[0.0  1.0]
```

```
>>> fp.pretty = False
>>> mp.pretty = False
```

Arbitrary-precision floating-point (mp)

The mp context is what most users probably want to use most of the time, as it supports the most functions, is most well-tested, and is implemented with a high level of optimization. Nearly all examples in this documentation use mp functions.

See *Basic usage* (page 623) for a description of basic usage.

Arbitrary-precision interval arithmetic (iv)

The iv.mpf type represents a closed interval $[a, b]$; that is, the set $\{x : a \leq x \leq b\}$, where a and b are arbitrary-precision floating-point values, possibly $\pm\infty$. The iv.mpc type represents a rectangular complex interval $[a, b] + [c, d]i$; that is, the set $\{z = x + iy : a \leq x \leq b \wedge c \leq y \leq d\}$.

Interval arithmetic provides rigorous error tracking. If f is a mathematical function and $\hat{f}$ is its interval arithmetic version, then the basic guarantee of interval arithmetic is that $f(v) \subseteq \hat{f}(v)$ for any input interval v. Put differently, if an interval represents the known uncertainty for a fixed number, any sequence of interval operations will produce an interval that contains what would be the result of applying the same sequence of operations to the exact number. The principal drawbacks of interval arithmetic are speed (iv arithmetic is typically at least two times slower than mp arithmetic) and that it sometimes provides far too pessimistic bounds.

Note: The support for interval arithmetic in mpmath is still experimental, and many functions do not yet properly support intervals. Please use this feature with caution.

Intervals can be created from single numbers (treated as zero-width intervals) or pairs of endpoint numbers. Strings are treated as exact decimal numbers. Note that a Python float like 0.1 generally does not represent the same number as its literal; use '0.1' instead:

```
>>> from mpmath import iv
>>> iv.dps = 15; iv.pretty = False
>>> iv.mpf(3)
mpi('3.0', '3.0')
>>> print iv.mpf(3)
[3.0, 3.0]
>>> iv.pretty = True
>>> iv.mpf([2,3])
[2.0, 3.0]
>>> iv.mpf(0.1)   # probably not intended
[0.10000000000000000555, 0.10000000000000000555]
>>> iv.mpf('0.1')   # good, gives a containing interval
[0.09999999999999991673, 0.10000000000000000555]
>>> iv.mpf(['0.1', '0.2'])
[0.09999999999999991673, 0.20000000000000000111]
```

The fact that '0.1' results in an interval of nonzero width indicates that 1/10 cannot be represented using binary floating-point numbers at this precision level (in fact, it cannot be represented exactly at any precision).

Intervals may be infinite or half-infinite:

```
>>> print 1 / iv.mpf([2, 'inf'])
[0.0, 0.5]
```

The equality testing operators `==` and `!=` check whether their operands are identical as intervals; that is, have the same endpoints. The ordering operators `< <= > >=` permit inequality testing using triple-valued logic: a guaranteed inequality returns `True` or `False` while an indeterminate inequality returns `None`:

```
>>> iv.mpf([1,2]) == iv.mpf([1,2])
True
>>> iv.mpf([1,2]) != iv.mpf([1,2])
False
>>> iv.mpf([1,2]) <= 2
True
>>> iv.mpf([1,2]) > 0
True
>>> iv.mpf([1,2]) < 1
False
>>> iv.mpf([1,2]) < 2    # returns None
>>> iv.mpf([2,2]) < 2
False
>>> iv.mpf([1,2]) <= iv.mpf([2,3])
True
>>> iv.mpf([1,2]) < iv.mpf([2,3])  # returns None
>>> iv.mpf([1,2]) < iv.mpf([-1,0])
False
```

The `in` operator tests whether a number or interval is contained in another interval:

```
>>> iv.mpf([0,2]) in iv.mpf([0,10])
True
>>> 3 in iv.mpf(['-inf', 0])
False
```

Intervals have the properties `.a`, `.b` (endpoints), `.mid`, and `.delta` (width):

```
>>> x = iv.mpf([2, 5])
>>> x.a
[2.0, 2.0]
>>> x.b
[5.0, 5.0]
>>> x.mid
[3.5, 3.5]
>>> x.delta
[3.0, 3.0]
```

Some transcendental functions are supported:

```
>>> iv.dps = 15
>>> mp.dps = 15
>>> iv.mpf([0.5,1.5]) ** iv.mpf([0.5, 1.5])
[0.35355339059327373086, 1.837117307087383633]
>>> iv.exp(0)
[1.0, 1.0]
>>> iv.exp(['-inf','inf'])
[0.0, +inf]
>>>
>>> iv.exp(['-inf',0])
[0.0, 1.0]
```

```
>>> iv.exp([0,'inf'])
[1.0, +inf]
>>> iv.exp([0,1])
[1.0, 2.7182818284590455349]
>>>
>>> iv.log(1)
[0.0, 0.0]
>>> iv.log([0,1])
[-inf, 0.0]
>>> iv.log([0,'inf'])
[-inf, +inf]
>>> iv.log(2)
[0.69314718055994528623, 0.69314718055994539725]
>>>
>>> iv.sin([100,'inf'])
[-1.0, 1.0]
>>> iv.cos(['-0.1','0.1'])
[0.99500416527802570954, 1.0]
```

Interval arithmetic is useful for proving inequalities involving irrational numbers. Naive use of `mp` arithmetic may result in wrong conclusions, such as the following:

```
>>> mp.dps = 25
>>> x = mp.exp(mp.pi*mp.sqrt(163))
>>> y = mp.mpf(640320**3+744)
>>> print x
262537412640768744.0000001
>>> print y
262537412640768744.0
>>> x > y
True
```

But the correct result is $e^{\pi\sqrt{163}} < 262537412640768744$, as can be seen by increasing the precision:

```
>>> mp.dps = 50
>>> print mp.exp(mp.pi*mp.sqrt(163))
262537412640768743.99999999999925007259719818568888
```

With interval arithmetic, the comparison returns `None` until the precision is large enough for $x - y$ to have a definite sign:

```
>>> iv.dps = 15
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
>>> iv.dps = 30
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
>>> iv.dps = 60
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
False
>>> iv.dps = 15
```

Fast low-precision arithmetic (`fp`)

Although mpmath is generally designed for arbitrary-precision arithmetic, many of the high-level algorithms work perfectly well with ordinary Python `float` and `complex` numbers, which use hardware double precision (on most systems, this corresponds to 53 bits of precision). Whereas the global functions (which are methods of the `mp` object) always convert inputs to mpmath numbers, the `fp` object instead converts them to `float` or `complex`, and in some

cases employs basic functions optimized for double precision. When large amounts of function evaluations (numerical integration, plotting, etc) are required, and when `fp` arithmetic provides sufficient accuracy, this can give a significant speedup over `mp` arithmetic.

To take advantage of this feature, simply use the `fp` prefix, i.e. write `fp.func` instead of `func` or `mp.func`:

```
>>> u = fp.erfc(2.5)
>>> print u
0.000406952017445
>>> type(u)
<type 'float'>
>>> mp.dps = 15
>>> print mp.erfc(2.5)
0.000406952017444959
>>> fp.matrix([[1,2],[3,4]]) ** 2
matrix(
[['7.0', '10.0'],
 ['15.0', '22.0']])
>>>
>>> type(_[0,0])
<type 'float'>
>>> print fp.quad(fp.sin, [0, fp.pi])    # numerical integration
2.0
```

The `fp` context wraps Python's `math` and `cmath` modules for elementary functions. It supports both real and complex numbers and automatically generates complex results for real inputs (`math` raises an exception):

```
>>> fp.sqrt(5)
2.2360679774997898
>>> fp.sqrt(-5)
2.2360679774997898j
>>> fp.sin(10)
-0.54402111088936977
>>> fp.power(-1, 0.25)
(0.70710678118654757+0.70710678118654746j)
>>> (-1) ** 0.25
Traceback (most recent call last):
  ...
ValueError: negative number cannot be raised to a fractional power
```

The `prec` and `dps` attributes can be changed (for interface compatibility with the `mp` context) but this has no effect:

```
>>> fp.prec
53
>>> fp.dps
15
>>> fp.prec = 80
>>> fp.prec
53
>>> fp.dps
15
```

Due to intermediate rounding and cancellation errors, results computed with `fp` arithmetic may be much less accurate than those computed with `mp` using an equivalent precision (`mp.prec = 53`), since the latter often uses increased internal precision. The accuracy is highly problem-dependent: for some functions, `fp` almost always gives 14-15 correct digits;

for others, results can be accurate to only 2-3 digits or even completely wrong. The recommended use for `fp` is therefore to speed up large-scale computations where accuracy can be verified in advance on a subset of the input set, or where results can be verified afterwards.

Utility functions

This page lists functions that perform basic operations on numbers or aid general programming.

Conversion and printing

mpmathify() / convert()

mpmath.**mpmathify**(*x*, *strings=True*)

Converts *x* to an `mpf` or `mpc`. If *x* is of type `mpf`, `mpc`, `int`, `float`, `complex`, the conversion will be performed losslessly.

If *x* is a string, the result will be rounded to the present working precision. Strings representing fractions or complex numbers are permitted.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> mpmathify(3.5)
mpf('3.5')
>>> mpmathify('2.1')
mpf('2.1000000000000001')
>>> mpmathify('3/4')
mpf('0.75')
>>> mpmathify('2+3j')
mpc(real='2.0', imag='3.0')
```

nstr()

mpmath.**nstr**(*x*, *n=6*, ***kwargs*)

Convert an `mpf` or `mpc` to a decimal string literal with *n* significant digits. The small default value for *n* is chosen to make this function useful for printing collections of numbers (lists, matrices, etc).

If *x* is a list or tuple, `nstr()` (page 634) is applied recursively to each element. For unrecognized classes, `nstr()` (page 634) simply returns `str(x)`.

The companion function `nprint()` (page 634) prints the result instead of returning it.

```
>>> from mpmath import *
>>> nstr([+pi, ldexp(1,-500)])
'[3.14159, 3.05494e-151]'
>>> nprint([+pi, ldexp(1,-500)])
[3.14159, 3.05494e-151]
```

nprint()

mpmath.**nprint**(*x*, *n=6*, ***kwargs*)

Equivalent to `print(nstr(x, n))`.

Arithmetic operations

See also `mpmath.sqrt()` (page 658), `mpmath.exp()` (page 662) etc., listed in *Powers and logarithms* (page 657)

fadd()

mpmath.**fadd**(*ctx*, *x*, *y*, ***kwargs*)

Adds the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

The default precision is the working precision of the context. You can specify a custom precision in bits by passing the *prec* keyword argument, or by providing an equivalent decimal precision with the *dps* keyword argument. If the precision is set to `+inf`, or if the flag *exact=True* is passed, an exact addition with no rounding is performed.

When the precision is finite, the optional *rounding* keyword argument specifies the direction of rounding. Valid options are `'n'` for nearest (default), `'f'` for floor, `'c'` for ceiling, `'d'` for down, `'u'` for up.

Examples

Using `fadd()` (page 635) with precision and rounding control:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fadd(2, 1e-20)
mpf('2.0')
>>> fadd(2, 1e-20, rounding='u')
mpf('2.0000000000000004')
>>> nprint(fadd(2, 1e-20, prec=100), 25)
2.00000000000000000001
>>> nprint(fadd(2, 1e-20, dps=15), 25)
2.0
>>> nprint(fadd(2, 1e-20, dps=25), 25)
2.00000000000000000001
>>> nprint(fadd(2, 1e-20, exact=True), 25)
2.00000000000000000001
```

Exact addition avoids cancellation errors, enforcing familiar laws of numbers such as $x + y - x = y$, which don't hold in floating-point arithmetic with finite precision:

```
>>> x, y = mpf(2), mpf('1e-1000')
>>> print(x + y - x)
0.0
>>> print(fadd(x, y, prec=inf) - x)
1.0e-1000
>>> print(fadd(x, y, exact=True) - x)
1.0e-1000
```

Exact addition can be inefficient and may be impossible to perform with large magnitude differences:

```
>>> fadd(1, '1e-1000000000000000000', prec=inf)
Traceback (most recent call last):
  ...
OverflowError: the exact result does not fit in memory
```

fsub()

mpmath.**fsub**(*ctx, x, y, **kwargs*)

Subtracts the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of fadd() (page 635) for a detailed description of how to specify precision and rounding.

Examples

Using fsub() (page 635) with precision and rounding control:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fsub(2, 1e-20)
mpf('2.0')
>>> fsub(2, 1e-20, rounding='d')
mpf('1.9999999999999998')
>>> nprint(fsub(2, 1e-20, prec=100), 25)
1.99999999999999999999
>>> nprint(fsub(2, 1e-20, dps=15), 25)
2.0
>>> nprint(fsub(2, 1e-20, dps=25), 25)
1.99999999999999999999
>>> nprint(fsub(2, 1e-20, exact=True), 25)
1.99999999999999999999
```

Exact subtraction avoids cancellation errors, enforcing familiar laws of numbers such as $x - y + y = x$, which don't hold in floating-point arithmetic with finite precision:

```
>>> x, y = mpf(2), mpf('1e1000')
>>> print(x - y + y)
0.0
>>> print(fsub(x, y, prec=inf) + y)
2.0
>>> print(fsub(x, y, exact=True) + y)
2.0
```

Exact addition can be inefficient and may be impossible to perform with large magnitude differences:

```
>>> fsub(1, '1e-10000000000000000000', prec=inf)
Traceback (most recent call last):
  ...
OverflowError: the exact result does not fit in memory
```

fneg()

mpmath.**fneg**(*ctx, x, **kwargs*)

Negates the number *x*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of fadd() (page 635) for a detailed description of how to specify precision and rounding.

Examples

An mpmath number is returned:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fneg(2.5)
```

```
mpf('-2.5')
>>> fneg(-5+2j)
mpc(real='5.0', imag='-2.0')
```

Precise control over rounding is possible:

```
>>> x = fadd(2, 1e-100, exact=True)
>>> fneg(x)
mpf('-2.0')
>>> fneg(x, rounding='f')
mpf('-2.0000000000000004')
```

Negating with and without roundoff:

```
>>> n = 200000000000000000000001
>>> print(int(-mpf(n)))
-200000000000000016777216
>>> print(int(fneg(n)))
-200000000000000016777216
>>> print(int(fneg(n, prec=log(n,2)+1)))
-200000000000000000000001
>>> print(int(fneg(n, dps=log(n,10)+1)))
-200000000000000000000001
>>> print(int(fneg(n, prec=inf)))
-200000000000000000000001
>>> print(int(fneg(n, dps=inf)))
-200000000000000000000001
>>> print(int(fneg(n, exact=True)))
-200000000000000000000001
```

fmul()

mpmath.**fmul**(*ctx, x, y, **kwargs*)

Multiplies the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` (page 635) for a detailed description of how to specify precision and rounding.

Examples

The result is an mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fmul(2, 5.0)
mpf('10.0')
>>> fmul(0.5j, 0.5)
mpc(real='0.0', imag='0.25')
```

Avoiding roundoff:

```
>>> x, y = 10**10+1, 10**15+1
>>> print(x*y)
10000000001000010000000001
>>> print(mpf(x) * mpf(y))
1.0000000001e+25
>>> print(int(mpf(x) * mpf(y)))
10000000001000011026399232
>>> print(int(fmul(x, y)))
```

```
1000000000100001102639923 2
>>> print(int(fmul(x, y, dps=25)))
1000000000100001000000001
>>> print(int(fmul(x, y, exact=True)))
1000000000100001000000001
```

Exact multiplication with complex numbers can be inefficient and may be impossible to perform with large magnitude differences between real and imaginary parts:

```
>>> x = 1+2j
>>> y = mpc(2, '1e-10000000000000000000')
>>> fmul(x, y)
mpc(real='2.0', imag='4.0')
>>> fmul(x, y, rounding='u')
mpc(real='2.0', imag='4.0000000000000009')
>>> fmul(x, y, exact=True)
Traceback (most recent call last):
  ...
OverflowError: the exact result does not fit in memory
```

fdiv()

mpmath.**fdiv**(*ctx, x, y, **kwargs*)

Divides the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` (page 635) for a detailed description of how to specify precision and rounding.

Examples

The result is an mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fdiv(3, 2)
mpf('1.5')
>>> fdiv(2, 3)
mpf('0.66666666666666663')
>>> fdiv(2+4j, 0.5)
mpc(real='4.0', imag='8.0')
```

The rounding direction and precision can be controlled:

```
>>> fdiv(2, 3, dps=3)    # Should be accurate to at least 3 digits
mpf('0.6666259765625')
>>> fdiv(2, 3, rounding='d')
mpf('0.66666666666666663')
>>> fdiv(2, 3, prec=60)
mpf('0.66666666666666667')
>>> fdiv(2, 3, rounding='u')
mpf('0.66666666666666674')
```

Checking the error of a division by performing it at higher precision:

```
>>> fdiv(2, 3) - fdiv(2, 3, prec=100)
mpf('-3.7007434154172148e-17')
```

Unlike `fadd()` (page 635), `fmul()` (page 637), etc., exact division is not allowed since the quotient of two floating-point numbers generally does not have an exact floating-point

representation. (In the future this might be changed to allow the case where the division is actually exact.)

```
>>> fdiv(2, 3, exact=True)
Traceback (most recent call last):
  ...
ValueError: division is not an exact operation
```

fmod()

mpmath.**fmod**(x, y)

Converts x and y to mpmath numbers and returns $x \mod y$. For mpmath numbers, this is equivalent to `x % y`.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> fmod(100, pi)
2.61062773871641
```

You can use `fmod()` (page 639) to compute fractional parts of numbers:

```
>>> fmod(10.25, 1)
0.25
```

fsum()

mpmath.**fsum**(*terms*, *absolute=False*, *squared=False*)

Calculates a sum containing a finite number of terms (for infinite series, see `nsum()` (page 882)). The terms will be converted to mpmath numbers. For len(terms) > 2, this function is generally faster and produces more accurate results than the builtin Python function `sum()`.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fsum([1, 2, 0.5, 7])
mpf('10.5')
```

With squared=True each term is squared, and with absolute=True the absolute value of each term is used.

fprod()

mpmath.**fprod**(*factors*)

Calculates a product containing a finite number of factors (for infinite products, see `nprod()` (page 889)). The factors will be converted to mpmath numbers.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fprod([1, 2, 0.5, 7])
mpf('7.0')
```

fdot()

mpmath.**fdot**(A, B=*None*, *conjugate=False*)

Computes the dot product of the iterables A and B,

$$\sum_{k=0} A_k B_k.$$

Alternatively, fdot() (page 639) accepts a single iterable of pairs. In other words, fdot(A,B) and fdot(zip(A,B)) are equivalent. The elements are automatically converted to mpmath numbers.

With conjugate=True, the elements in the second vector will be conjugated:

$$\sum_{k=0} A_k \overline{B_k}$$

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> A = [2, 1.5, 3]
>>> B = [1, -1, 2]
>>> fdot(A, B)
mpf('6.5')
>>> list(zip(A, B))
[(2, 1), (1.5, -1), (3, 2)]
>>> fdot(_)
mpf('6.5')
>>> A = [2, 1.5, 3j]
>>> B = [1+j, 3, -1-j]
>>> fdot(A, B)
mpc(real='9.5', imag='-1.0')
>>> fdot(A, B, conjugate=True)
mpc(real='3.5', imag='-5.0')
```

Complex components

fabs()

mpmath.**fabs**(x)

Returns the absolute value of x, $|x|$. Unlike abs(), fabs() (page 640) converts non-mpmath numbers (such as int) into mpmath numbers:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> fabs(3)
mpf('3.0')
>>> fabs(-3)
mpf('3.0')
>>> fabs(3+4j)
mpf('5.0')
```

sign()

mpmath.**sign**(x)

Returns the sign of x, defined as $\operatorname{sign}(x) = x/|x|$ (with the special case $\operatorname{sign}(0) = 0$):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> sign(10)
mpf('1.0')
>>> sign(-10)
mpf('-1.0')
>>> sign(0)
mpf('0.0')
```

Note that the sign function is also defined for complex numbers, for which it gives the projection onto the unit circle:

```
>>> mp.dps = 15; mp.pretty = True
>>> sign(1+j)
(0.707106781186547 + 0.707106781186547j)
```

re()

mpmath.**re**(x)

Returns the real part of x, $\Re(x)$. Unlike `x.real`, `re()` (page 641) converts x to a mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> re(3)
mpf('3.0')
>>> re(-1+4j)
mpf('-1.0')
```

im()

mpmath.**im**(x)

Returns the imaginary part of x, $\Im(x)$. Unlike `x.imag`, `im()` (page 641) converts x to a mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> im(3)
mpf('0.0')
>>> im(-1+4j)
mpf('4.0')
```

arg()

mpmath.**arg**(x)

Computes the complex argument (phase) of x, defined as the signed angle between the positive real axis and x in the complex plane:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> arg(3)
0.0
>>> arg(3+3j)
0.785398163397448
>>> arg(3j)
1.5707963267949
>>> arg(-3)
3.14159265358979
>>> arg(-3j)
-1.5707963267949
```

The angle is defined to satisfy $-\pi < \arg(x) \leq \pi$ and with the sign convention that a nonnegative imaginary part results in a nonnegative argument.

The value returned by `arg()` (page 641) is an `mpf` instance.

conj()

mpmath.**conj**(x)

Returns the complex conjugate of x, $\overline{x}$. Unlike `x.conjugate()`, `im()` (page 641) converts x to a mpmath number:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> conj(3)
mpf('3.0')
>>> conj(-1+4j)
mpc(real='-1.0', imag='-4.0')
```

polar()

mpmath.**polar**(x)

Returns the polar representation of the complex number z as a pair (r, ϕ) such that $z = re^{i\phi}$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> polar(-2)
(2.0, 3.14159265358979)
>>> polar(3-4j)
(5.0, -0.927295218001612)
```

rect()

mpmath.**rect**(x)

Returns the complex number represented by polar coordinates (r, ϕ):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> chop(rect(2, pi))
-2.0
>>> rect(sqrt(2), -pi/4)
(1.0 - 1.0j)
```

Integer and fractional parts

floor()

mpmath.**floor**(x)

Computes the floor of x, $\lfloor x \rfloor$, defined as the largest integer less than or equal to x:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> floor(3.5)
mpf('3.0')
```

Note: `floor()` (page 642), `ceil()` (page 643) and `nint()` (page 643) return a floating-point number, not a Python `int`. If $\lfloor x \rfloor$ is too large to be represented exactly at the present working precision, the result will be rounded, not necessarily in the direction implied by the mathematical definition of the function.

To avoid rounding, use *prec=0*:

```
>>> mp.dps = 15
>>> print(int(floor(10**30+1)))
1000000000000000019884624838656
>>> print(int(floor(10**30+1, prec=0)))
1000000000000000000000000000001
```

The floor function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> floor(3.25+4.75j)
mpc(real='3.0', imag='4.0')
```

ceil()

mpmath.**ceil**(x)

Computes the ceiling of x, $\lceil x \rceil$, defined as the smallest integer greater than or equal to x:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> ceil(3.5)
mpf('4.0')
```

The ceiling function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> ceil(3.25+4.75j)
mpc(real='4.0', imag='5.0')
```

See notes about rounding for `floor()` (page 642).

nint()

mpmath.**nint**(x)

Evaluates the nearest integer function, $\mathrm{nint}(x)$. This gives the nearest integer to x; on a tie, it gives the nearest even integer:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> nint(3.2)
mpf('3.0')
>>> nint(3.8)
mpf('4.0')
>>> nint(3.5)
mpf('4.0')
>>> nint(4.5)
mpf('4.0')
```

The nearest integer function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> nint(3.25+4.75j)
mpc(real='3.0', imag='5.0')
```

See notes about rounding for `floor()` (page 642).

frac()

mpmath.**frac**(*x*)

Gives the fractional part of x, defined as $\text{frac}(x) = x - \lfloor x \rfloor$ (see `floor()` (page 642)). In effect, this computes x modulo 1, or $x + n$ where $n \in \mathbb{Z}$ is such that $x + n \in [0, 1)$:

```
>>> from mpmath import *
>>> mp.pretty = False
>>> frac(1.25)
mpf('0.25')
>>> frac(3)
mpf('0.0')
>>> frac(-1.25)
mpf('0.75')
```

For a complex number, the fractional part function applies to the real and imaginary parts separately:

```
>>> frac(2.25+3.75j)
mpc(real='0.25', imag='0.75')
```

Plotted, the fractional part function gives a sawtooth wave. The Fourier series coefficients have a simple form:

```
>>> mp.dps = 15
>>> nprint(fourier(lambda x: frac(x)-0.5, [0,1], 4))
([0.0, 0.0, 0.0, 0.0, 0.0], [0.0, -0.31831, -0.159155, -0.106103, -0.0795775])
>>> nprint([-1/(pi*k) for k in range(1,5)])
[-0.31831, -0.159155, -0.106103, -0.0795775]
```

Note: The fractional part is sometimes defined as a symmetric function, i.e. returning $-\text{frac}(-x)$ if $x < 0$. This convention is used, for instance, by Mathematica's `FractionalPart`.

Tolerances and approximate comparisons

chop()

mpmath.**chop**(*x*, *tol=None*)

Chops off small real or imaginary parts, or converts numbers close to zero to exact zeros. The input can be a single number or an iterable:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> chop(5+1e-10j, tol=1e-9)
mpf('5.0')
>>> nprint(chop([1.0, 1e-20, 3+1e-18j, -4, 2]))
[1.0, 0.0, 3.0, -4.0, 2.0]
```

The tolerance defaults to `100*eps`.

almosteq()

mpmath.**almosteq**(*s*, *t*, *rel_eps=None*, *abs_eps=None*)

Determine whether the difference between s and t is smaller than a given epsilon, either relatively or absolutely.

Both a maximum relative difference and a maximum difference ('epsilons') may be specified. The absolute difference is defined as $|s-t|$ and the relative difference is defined as $|s-t|/\max(|s|,|t|)$.

If only one epsilon is given, both are set to the same value. If none is given, both epsilons are set to 2^{-p+m} where p is the current working precision and m is a small integer. The default setting typically allows `almosteq()` (page 644) to be used to check for mathematical equality in the presence of small rounding errors.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15
>>> almosteq(3.141592653589793, 3.141592653589790)
True
>>> almosteq(3.141592653589793, 3.141592653589700)
False
>>> almosteq(3.141592653589793, 3.141592653589700, 1e-10)
True
>>> almosteq(1e-20, 2e-20)
True
>>> almosteq(1e-20, 2e-20, rel_eps=0, abs_eps=0)
False
```

Properties of numbers

isinf()

mpmath.**isinf**(*x*)

Return *True* if the absolute value of *x* is infinite; otherwise return *False*:

```
>>> from mpmath import *
>>> isinf(inf)
True
>>> isinf(-inf)
True
>>> isinf(3)
False
>>> isinf(3+4j)
False
>>> isinf(mpc(3,inf))
True
>>> isinf(mpc(inf,3))
True
```

isnan()

mpmath.**isnan**(*x*)

Return *True* if *x* is a NaN (not-a-number), or for a complex number, whether either the real or complex part is NaN; otherwise return *False*:

```
>>> from mpmath import *
>>> isnan(3.14)
False
>>> isnan(nan)
True
>>> isnan(mpc(3.14,2.72))
False
```

```
>>> isnan(mpc(3.14,nan))
True
```

isnormal()

mpmath.**isnormal**(*x*)

Determine whether *x* is "normal" in the sense of floating-point representation; that is, return *False* if *x* is zero, an infinity or NaN; otherwise return *True*. By extension, a complex number *x* is considered "normal" if its magnitude is normal:

```
>>> from mpmath import *
>>> isnormal(3)
True
>>> isnormal(0)
False
>>> isnormal(inf); isnormal(-inf); isnormal(nan)
False
False
False
>>> isnormal(0+0j)
False
>>> isnormal(0+3j)
True
>>> isnormal(mpc(2,nan))
False
```

isint()

mpmath.**isint**(*x*, *gaussian=False*)

Return *True* if *x* is integer-valued; otherwise return *False*:

```
>>> from mpmath import *
>>> isint(3)
True
>>> isint(mpf(3))
True
>>> isint(3.2)
False
>>> isint(inf)
False
```

Optionally, Gaussian integers can be checked for:

```
>>> isint(3+0j)
True
>>> isint(3+2j)
False
>>> isint(3+2j, gaussian=True)
True
```

ldexp()

mpmath.**ldexp**(*x*, *n*)

Computes $x2^n$ efficiently. No rounding is performed. The argument x must be a real floating-point number (or possible to convert into one) and n must be a Python `int`.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> ldexp(1, 10)
mpf('1024.0')
>>> ldexp(1, -3)
mpf('0.125')
```

frexp()

mpmath.**frexp**(*x*, *n*)

Given a real number x, returns (y, n) with $y \in [0.5, 1)$, n a Python integer, and such that $x = y2^n$. No rounding is performed.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> frexp(7.5)
(mpf('0.9375'), 3)
```

mag()

mpmath.**mag**(*x*)

Quick logarithmic magnitude estimate of a number. Returns an integer or infinity m such that $|x| <= 2^m$. It is not guaranteed that m is an optimal bound, but it will never be too large by more than 2 (and probably not more than 1).

Examples

```
>>> from mpmath import *
>>> mp.pretty = True
>>> mag(10), mag(10.0), mag(mpf(10)), int(ceil(log(10,2)))
(4, 4, 4, 4)
>>> mag(10j), mag(10+10j)
(4, 5)
>>> mag(0.01), int(ceil(log(0.01,2)))
(-6, -6)
>>> mag(0), mag(inf), mag(-inf), mag(nan)
(-inf, +inf, +inf, nan)
```

nint_distance()

mpmath.**nint_distance**(*x*)

Return (n, d) where n is the nearest integer to x and d is an estimate of $\log_2(|x - n|)$. If $d < 0$, $-d$ gives the precision (measured in bits) lost to cancellation when computing $x - n$.

```
>>> from mpmath import *
>>> n, d = nint_distance(5)
>>> print(n); print(d)
5
-inf
>>> n, d = nint_distance(mpf(5))
>>> print(n); print(d)
5
-inf
>>> n, d = nint_distance(mpf(5.00000001))
>>> print(n); print(d)
5
-26
```

```
>>> n, d = nint_distance(mpf(4.99999999))
>>> print(n); print(d)
5
-26
>>> n, d = nint_distance(mpc(5,10))
>>> print(n); print(d)
5
4
>>> n, d = nint_distance(mpc(5,0.000001))
>>> print(n); print(d)
5
-19
```

Number generation

fraction()

mpmath.**fraction**(*p, q*)

Given Python integers (p, q), returns a lazy `mpf` representing the fraction p/q. The value is updated with the precision.

```
>>> from mpmath import *
>>> mp.dps = 15
>>> a = fraction(1,100)
>>> b = mpf(1)/100
>>> print(a); print(b)
0.01
0.01
>>> mp.dps = 30
>>> print(a); print(b)      # a will be accurate
0.01
0.0100000000000000000020816681711 72
>>> mp.dps = 15
```

rand()

mpmath.**rand**()

Returns an `mpf` with value chosen randomly from $[0, 1)$. The number of randomly generated bits in the mantissa is equal to the working precision.

arange()

mpmath.**arange**(**args*)

This is a generalized version of Python's `range()` function that accepts fractional endpoints and step sizes and returns a list of `mpf` instances. Like `range()`, `arange()` (page 648) can be called with 1, 2 or 3 arguments:

arange(b) $[0, 1, 2, \ldots, x]$

arange(a, b) $[a, a+1, a+2, \ldots, x]$

arange(a, b, h) $[a, a+h, a+h, \ldots, x]$

where $b - 1 \leq x < b$ (in the third case, $b - h \leq x < b$).

Like Python's `range()`, the endpoint is not included. To produce ranges where the endpoint is included, `linspace()` (page 649) is more convenient.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> arange(4)
[mpf('0.0'), mpf('1.0'), mpf('2.0'), mpf('3.0')]
>>> arange(1, 2, 0.25)
[mpf('1.0'), mpf('1.25'), mpf('1.5'), mpf('1.75')]
>>> arange(1, -1, -0.75)
[mpf('1.0'), mpf('0.25'), mpf('-0.5')]
```

linspace()

mpmath.**linspace**(**args, **kwargs*)

`linspace(a, b, n)` returns a list of n evenly spaced samples from a to b. The syntax `linspace(mpi(a,b), n)` is also valid.

This function is often more convenient than `arange()` (page 648) for partitioning an interval into subintervals, since the endpoint is included:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> linspace(1, 4, 4)
[mpf('1.0'), mpf('2.0'), mpf('3.0'), mpf('4.0')]
```

You may also provide the keyword argument `endpoint=False`:

```
>>> linspace(1, 4, 4, endpoint=False)
[mpf('1.0'), mpf('1.75'), mpf('2.5'), mpf('3.25')]
```

Precision management

autoprec()

mpmath.**autoprec**(*ctx, f, maxprec=None, catch=(), verbose=False*)

Return a wrapped copy of f that repeatedly evaluates f with increasing precision until the result converges to the full precision used at the point of the call.

This heuristically protects against rounding errors, at the cost of roughly a 2x slowdown compared to manually setting the optimal precision. This method can, however, easily be fooled if the results from f depend "discontinuously" on the precision, for instance if catastrophic cancellation can occur. Therefore, `autoprec()` (page 649) should be used judiciously.

Examples

Many functions are sensitive to perturbations of the input arguments. If the arguments are decimal numbers, they may have to be converted to binary at a much higher precision. If the amount of required extra precision is unknown, `autoprec()` (page 649) is convenient:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> mp.pretty = True
>>> besselj(5, 125 * 10**28)    # Exact input
-8.03284785591801e-17
>>> besselj(5, '1.25e30')   # Bad
7.12954868316652e-16
>>> autoprec(besselj)(5, '1.25e30')   # Good
-8.03284785591801e-17
```

The following fails to converge because $\sin(\pi) = 0$ whereas all finite-precision approximations of π give nonzero values:

```
>>> autoprec(sin)(pi)
Traceback (most recent call last):
  ...
NoConvergence: autoprec: prec increased to 2910 without convergence
```

As the following example shows, `autoprec()` (page 649) can protect against cancellation, but is fooled by too severe cancellation:

```
>>> x = 1e-10
>>> exp(x)-1; expm1(x); autoprec(lambda t: exp(t)-1)(x)
1.00000008274037e-10
1.00000000005e-10
1.00000000005e-10
>>> x = 1e-50
>>> exp(x)-1; expm1(x); autoprec(lambda t: exp(t)-1)(x)
0.0
1.0e-50
0.0
```

With *catch*, an exception or list of exceptions to intercept may be specified. The raised exception is interpreted as signaling insufficient precision. This permits, for example, evaluating a function where a too low precision results in a division by zero:

```
>>> f = lambda x: 1/(exp(x)-1)
>>> f(1e-30)
Traceback (most recent call last):
  ...
ZeroDivisionError
>>> autoprec(f, catch=ZeroDivisionError)(1e-30)
1.0e+30
```

workprec()

mpmath.**workprec**(*ctx*, *n*, *normalize_output=False*)

The block

with workprec(n): <code>

sets the precision to n bits, executes <code>, and then restores the precision.

workprec(n)(f) returns a decorated version of the function f that sets the precision to n bits before execution, and restores the precision afterwards. With normalize_output=True, it rounds the return value to the parent precision.

workdps()

mpmath.**workdps**(*ctx*, *n*, *normalize_output=False*)

This function is analogous to workprec (see documentation) but changes the decimal precision instead of the number of bits.

extraprec()

mpmath.**extraprec**(*ctx*, *n*, *normalize_output=False*)

The block

with extraprec(n): <code>

increases the precision n bits, executes <code>, and then restores the precision.

extraprec(n)(f) returns a decorated version of the function f that increases the working precision by n bits before execution, and restores the parent precision afterwards. With normalize_output=True, it rounds the return value to the parent precision.

extradps()

mpmath.**extradps**(*ctx*, *n*, *normalize_output=False*)

This function is analogous to extraprec (see documentation) but changes the decimal precision instead of the number of bits.

Performance and debugging

memoize()

mpmath.**memoize**(*ctx*, *f*)

Return a wrapped copy of *f* that caches computed values, i.e. a memoized copy of *f*. Values are only reused if the cached precision is equal to or higher than the working precision:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = memoize(maxcalls(sin, 1))
>>> f(2)
0.909297426825682
>>> f(2)
0.909297426825682
>>> mp.dps = 25
>>> f(2)
Traceback (most recent call last):
  ...
NoConvergence: maxcalls: function evaluated 1 times
```

maxcalls()

mpmath.**maxcalls**(*ctx*, *f*, *N*)

Return a wrapped copy of *f* that raises `NoConvergence` when *f* has been called more than *N* times:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> f = maxcalls(sin, 10)
>>> print(sum(f(n) for n in range(10)))
1.95520948210738
>>> f(10)
Traceback (most recent call last):
  ...
NoConvergence: maxcalls: function evaluated 10 times
```

monitor()

mpmath.**monitor**(*f*, *input='print'*, *output='print'*)

Returns a wrapped copy of *f* that monitors evaluation by calling *input* with every input (*args*, *kwargs*) passed to *f* and *output* with every value returned from *f*. The default action (specify using the special string value `'print'`) is to print inputs and outputs to stdout, along with the total evaluation count:

```
>>> from mpmath import *
>>> mp.dps = 5; mp.pretty = False
>>> diff(monitor(exp), 1)   # diff will eval f(x-h) and f(x+h)
in  0 (mpf('0.99999999906867742538452148'),) {}
out 0 mpf('2.7182818259274480055282064')
in  1 (mpf('1.0000000009313225746154785'),) {}
out 1 mpf('2.7182818309906424675501024')
mpf('2.7182808')
```

To disable either the input or the output handler, you may pass *None* as argument.

Custom input and output handlers may be used e.g. to store results for later analysis:

```
>>> mp.dps = 15
>>> input = []
>>> output = []
>>> findroot(monitor(sin, input.append, output.append), 3.0)
mpf('3.1415926535897932')
>>> len(input)  # Count number of evaluations
9
>>> print(input[3]); print(output[3])
((mpf('3.1415076583334066'),), {})
8.49952562843408e-5
>>> print(input[4]); print(output[4])
((mpf('3.1415928201669122'),), {})
-1.66577118985331e-7
```

timing()

mpmath.**timing**(*f*, **args*, ***kwargs*)

Returns time elapsed for evaluating `f()`. Optionally arguments may be passed to time the execution of `f(*args, **kwargs)`.

If the first call is very quick, `f` is called repeatedly and the best time is returned.

Plotting

If matplotlib is available, the functions `plot` and `cplot` in mpmath can be used to plot functions respectively as x-y graphs and in the complex plane. Also, `splot` can be used to produce 3D surface plots.

Function curve plots

Output of `plot([cos, sin], [-4, 4])`

`mpmath.`**`plot`**(*ctx, f, xlim=[-5, 5], ylim=None, points=200, file=None, dpi=None, singularities=[], axes=None*)

Shows a simple 2D plot of a function $f(x)$ or list of functions $[f_0(x), f_1(x), \ldots, f_n(x)]$ over a given interval specified by *xlim*. Some examples:

```
plot(lambda x: exp(x)*li(x), [1, 4])
plot([cos, sin], [-4, 4])
plot([fresnels, fresnelc], [-4, 4])
plot([sqrt, cbrt], [-4, 4])
plot(lambda t: zeta(0.5+t*j), [-20, 20])
plot([floor, ceil, abs, sign], [-5, 5])
```

Points where the function raises a numerical exception or returns an infinite value are removed from the graph. Singularities can also be excluded explicitly as follows (useful for removing erroneous vertical lines):

```
plot(cot, ylim=[-5, 5])   # bad
plot(cot, ylim=[-5, 5], singularities=[-pi, 0, pi])  # good
```

For parts where the function assumes complex values, the real part is plotted with dashes and the imaginary part is plotted with dots.

Note: This function requires matplotlib (pylab).

Complex function plots

Output of `fp.cplot(fp.gamma, points=100000)`

`mpmath.`**`cplot`**(*ctx, f, re=[-5, 5], im=[-5, 5], points=2000, color=None, verbose=False, file=None, dpi=None, axes=None*)

Plots the given complex-valued function *f* over a rectangular part of the complex plane specified by the pairs of intervals *re* and *im*. For example:

```
cplot(lambda z: z, [-2, 2], [-10, 10])
cplot(exp)
cplot(zeta, [0, 1], [0, 50])
```

By default, the complex argument (phase) is shown as color (hue) and the magnitude is show as brightness. You can also supply a custom color function (*color*). This function should take a complex number as input and return an RGB 3-tuple containing floats in the range 0.0-1.0.

To obtain a sharp image, the number of points may need to be increased to 100,000 or thereabout. Since evaluating the function that many times is likely to be slow, the 'verbose' option is useful to display progress.

Note: This function requires matplotlib (pylab).

3D surface plots

Output of `splot` for the donut example.

`mpmath.`**`splot`**(*ctx, f, u=[-5, 5], v=[-5, 5], points=100, keep_aspect=True, wireframe=False, file=None, dpi=None, axes=None*)

Plots the surface defined by f.

If f returns a single component, then this plots the surface defined by $z = f(x, y)$ over the rectangular domain with $x = u$ and $y = v$.

If f returns three components, then this plots the parametric surface $x, y, z = f(u, v)$ over the pairs of intervals u and v.

For example, to plot a simple function:

```
>>> from mpmath import *
>>> f = lambda x, y: sin(x+y)*cos(y)
>>> splot(f, [-pi,pi], [-pi,pi])
```

Plotting a donut:

```
>>> r, R = 1, 2.5
>>> f = lambda u, v: [r*cos(u), (R+r*sin(u))*cos(v), (R+r*sin(u))*sin(v)]
>>> splot(f, [0, 2*pi], [0, 2*pi])
```

Note: This function requires matplotlib (pylab) 0.98.5.3 or higher.

5.15.3 Advanced mathematics

On top of its support for arbitrary-precision arithmetic, mpmath provides extensive support for transcendental functions, evaluation of sums, integrals, limits, roots, and so on.

Mathematical functions

Mpmath implements the standard functions from Python's `math` and `cmath` modules, for both real and complex numbers and with arbitrary precision. Many other functions are also available in mpmath, including commonly-used variants of standard functions (such as the alternative trigonometric functions sec, csc, cot), but also a large number of "special functions" such as the gamma function, the Riemann zeta function, error functions, Bessel functions, etc.

Mathematical constants

Mpmath supports arbitrary-precision computation of various common (and less common) mathematical constants. These constants are implemented as lazy objects that can evaluate to any precision. Whenever the objects are used as function arguments or as operands in arithmetic operations, they automagically evaluate to the current working precision. A lazy number can be converted to a regular `mpf` using the unary + operator, or by calling it as a function:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.2831853071795862')
>>> +pi
mpf('3.1415926535897931')
>>> pi()
mpf('3.1415926535897931')
>>> mp.dps = 40
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.2831853071795864769252867665590057683943387')
>>> +pi
mpf('3.1415926535897932384626433832795028841971693')
>>> pi()
mpf('3.1415926535897932384626433832795028841971693')
```

Exact constants The predefined objects `j` (imaginary unit), `inf` (positive infinity) and `nan` (not-a-number) are shortcuts to `mpc` and `mpf` instances with these fixed values.

Pi (`pi`)
`mp.pi` = <pi: 3.14159~>

Degree (`degree`)
`mp.degree` = <1 deg = pi / 180: 0.0174533~>

Base of the natural logarithm (`e`)
`mp.e` = <e = exp(1): 2.71828~>

Golden ratio (`phi`)
`mp.phi` = <Golden ratio phi: 1.61803~>

Euler's constant (`euler`)
`mp.euler` = <Euler's constant: 0.577216~>

Catalan's constant (`catalan`)
`mp.catalan` = <Catalan's constant: 0.915966~>

Apery's constant (`apery`)
`mp.apery` = <Apery's constant: 1.20206~>

Khinchin's constant (`khinchin`)
`mp.khinchin` = <Khinchin's constant: 2.68545~>

Glaisher's constant (`glaisher`)
`mp.glaisher` = <Glaisher's constant: 1.28243~>

Mertens constant (`mertens`)
`mp.mertens` = <Mertens' constant: 0.261497~>

Twin prime constant (`twinprime`)
`mp.twinprime` = <Twin prime constant: 0.660162~>

Powers and logarithms

Nth roots

sqrt()

mpmath.**sqrt**(*x*, ***kwargs*)

`sqrt(x)` gives the principal square root of x, $\sqrt{x}$. For positive real numbers, the principal root is simply the positive square root. For arbitrary complex numbers, the principal square root is defined to satisfy $\sqrt{x} = \exp(\log(x)/2)$. The function thus has a branch cut along the negative half real axis.

For all mpmath numbers `x`, calling `sqrt(x)` is equivalent to performing `x**0.5`.

Examples

Basic examples and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sqrt(10)
3.16227766016838
>>> sqrt(100)
10.0
>>> sqrt(-4)
(0.0 + 2.0j)
>>> sqrt(1+1j)
(1.09868411346781 + 0.455089860562227j)
>>> sqrt(inf)
+inf
```

Square root evaluation is fast at huge precision:

```
>>> mp.dps = 50000
>>> a = sqrt(3)
>>> str(a)[-10:]
'9329332814'
```

`mpmath.iv.sqrt()` supports interval arguments:

```
>>> iv.dps = 15; iv.pretty = True
>>> iv.sqrt([16,100])
[4.0, 10.0]
>>> iv.sqrt(2)
[1.4142135623730949234, 1.4142135623730951455]
>>> iv.sqrt(2) ** 2
[1.9999999999999995559, 2.0000000000000004441]
```

hypot()

mpmath.**hypot**(*x*, *y*)

Computes the Euclidean norm of the vector (x, y), equal to $\sqrt{x^2 + y^2}$. Both x and y must be real.

cbrt()

mpmath.**cbrt**(*x*, ***kwargs*)

`cbrt(x)` computes the cube root of x, $x^{1/3}$. This function is faster and more accurate than raising to a floating-point fraction:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> 125**(mpf(1)/3)
mpf('4.9999999999999991')
```

```
>>> cbrt(125)
mpf('5.0')
```

Every nonzero complex number has three cube roots. This function returns the cube root defined by $\exp(\log(x)/3)$ where the principal branch of the natural logarithm is used. Note that this does not give a real cube root for negative real numbers:

```
>>> mp.pretty = True
>>> cbrt(-1)
(0.5 + 0.866025403784439j)
```

root()

mpmath.**root**(*z, n, k=0*)

`root(z, n, k=0)` computes an n-th root of z, i.e. returns a number r that (up to possible approximation error) satisfies $r^n = z$. (`nthroot` is available as an alias for `root`.)

Every complex number $z \neq 0$ has n distinct n-th roots, which are equidistant points on a circle with radius $|z|^{1/n}$, centered around the origin. A specific root may be selected using the optional index k. The roots are indexed counterclockwise, starting with $k = 0$ for the root closest to the positive real half-axis.

The $k = 0$ root is the so-called principal n-th root, often denoted by $\sqrt[n]{z}$ or $z^{1/n}$, and also given by $\exp(\log(z)/n)$. If z is a positive real number, the principal root is just the unique positive n-th root of z. Under some circumstances, non-principal real roots exist: for positive real z, n even, there is a negative root given by $k = n/2$; for negative real z, n odd, there is a negative root given by $k = (n-1)/2$.

To obtain all roots with a simple expression, use `[root(z,n,k) for k in range(n)]`.

An important special case, `root(1, n, k)` returns the k-th n-th root of unity, $\zeta_k = e^{2\pi i k/n}$. Alternatively, `unitroots()` (page 660) provides a slightly more convenient way to obtain the roots of unity, including the option to compute only the primitive roots of unity.

Both k and n should be integers; k outside of `range(n)` will be reduced modulo n. If n is negative, $x^{-1/n} = 1/x^{1/n}$ (or the equivalent reciprocal for a non-principal root with $k \neq 0$) is computed.

`root()` (page 659) is implemented to use Newton's method for small n. At high precision, this makes $x^{1/n}$ not much more expensive than the regular exponentiation, x^n. For very large n, `nthroot()` falls back to use the exponential function.

Examples

`nthroot()`/`root()` (page 659) is faster and more accurate than raising to a floating-point fraction:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> 16807 ** (mpf(1)/5)
mpf('7.0000000000000009')
>>> root(16807, 5)
mpf('7.0')
>>> nthroot(16807, 5)    # Alias
mpf('7.0')
```

A high-precision root:

```
>>> mp.dps = 50; mp.pretty = True
>>> nthroot(10, 5)
```

```
1.5848931924611134852021013733915070132694421338 25
>>> nthroot(10, 5) ** 5
10.0
```

Computing principal and non-principal square and cube roots:

```
>>> mp.dps = 15
>>> root(10, 2)
3.16227766016838
>>> root(10, 2, 1)
-3.16227766016838
>>> root(-10, 3)
(1.07721734501594 + 1.86579517236206j)
>>> root(-10, 3, 1)
-2.15443469003188
>>> root(-10, 3, 2)
(1.07721734501594 - 1.86579517236206j)
```

All the 7th roots of a complex number:

```
>>> for r in [root(3+4j, 7, k) for k in range(7)]:
...     print("%s %s" % (r, r**7))
...
(1.24747270589553 + 0.166227124177353j) (3.0 + 4.0j)
(0.647824911301003 + 1.07895435170559j) (3.0 + 4.0j)
(-0.439648254723098 + 1.17920694574172j) (3.0 + 4.0j)
(-1.19605731775069 + 0.391492658196305j) (3.0 + 4.0j)
(-1.05181082538903 - 0.691023585965793j) (3.0 + 4.0j)
(-0.115529328478668 - 1.25318497558335j) (3.0 + 4.0j)
(0.907748109144957 - 0.871672518271819j) (3.0 + 4.0j)
```

Cube roots of unity:

```
>>> for k in range(3): print(root(1, 3, k))
...
1.0
(-0.5 + 0.866025403784439j)
(-0.5 - 0.866025403784439j)
```

Some exact high order roots:

```
>>> root(75**210, 105)
5625.0
>>> root(1, 128, 96)
(0.0 - 1.0j)
>>> root(4**128, 128, 96)
(0.0 - 4.0j)
```

unitroots()

mpmath.**unitroots**(*n*, *primitive=False*)

`unitroots(n)` returns $\zeta_0, \zeta_1, \ldots, \zeta_{n-1}$, all the distinct n-th roots of unity, as a list. If the option *primitive=True* is passed, only the primitive roots are returned.

Every n-th root of unity satisfies $(\zeta_k)^n = 1$. There are n distinct roots for each n (ζ_k and ζ_j are the same when $k = j \pmod{n}$), which form a regular polygon with vertices on the unit circle. They are ordered counterclockwise with increasing k, starting with $\zeta_0 = 1$.

Examples

The roots of unity up to $n = 4$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(unitroots(1))
[1.0]
>>> nprint(unitroots(2))
[1.0, -1.0]
>>> nprint(unitroots(3))
[1.0, (-0.5 + 0.866025j), (-0.5 - 0.866025j)]
>>> nprint(unitroots(4))
[1.0, (0.0 + 1.0j), -1.0, (0.0 - 1.0j)]
```

Roots of unity form a geometric series that sums to 0:

```
>>> mp.dps = 50
>>> chop(fsum(unitroots(25)))
0.0
```

Primitive roots up to $n = 4$:

```
>>> mp.dps = 15
>>> nprint(unitroots(1, primitive=True))
[1.0]
>>> nprint(unitroots(2, primitive=True))
[-1.0]
>>> nprint(unitroots(3, primitive=True))
[(-0.5 + 0.866025j), (-0.5 - 0.866025j)]
>>> nprint(unitroots(4, primitive=True))
[(0.0 + 1.0j), (0.0 - 1.0j)]
```

There are only four primitive 12th roots:

```
>>> nprint(unitroots(12, primitive=True))
[(0.866025 + 0.5j), (-0.866025 + 0.5j), (-0.866025 - 0.5j), (0.866025 - 0.5j)]
```

The n-th roots of unity form a group, the cyclic group of order n. Any primitive root r is a generator for this group, meaning that $r^0, r^1, \ldots, r^{n-1}$ gives the whole set of unit roots (in some permuted order):

```
>>> for r in unitroots(6): print(r)
...
1.0
(0.5 + 0.866025403784439j)
(-0.5 + 0.866025403784439j)
-1.0
(-0.5 - 0.866025403784439j)
(0.5 - 0.866025403784439j)
>>> r = unitroots(6, primitive=True)[1]
>>> for k in range(6): print(chop(r**k))
...
1.0
(0.5 - 0.866025403784439j)
(-0.5 - 0.866025403784439j)
-1.0
(-0.5 + 0.866025403784438j)
(0.5 + 0.866025403784438j)
```

The number of primitive roots equals the Euler totient function $\phi(n)$:

```
>>> [len(unitroots(n, primitive=True)) for n in range(1,20)]
[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]
```

Exponentiation

exp()

mpmath.**exp**(*x*, ***kwargs*)

Computes the exponential function,

$$\exp(x) = e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

For complex numbers, the exponential function also satisfies

$$\exp(x + yi) = e^x(\cos y + i \sin y).$$

Basic examples

Some values of the exponential function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> exp(0)
1.0
>>> exp(1)
2.718281828459045235360287
>>> exp(-1)
0.3678794411714423215955238
>>> exp(inf)
+inf
>>> exp(-inf)
0.0
```

Arguments can be arbitrarily large:

```
>>> exp(10000)
8.806818225662921587261496e+4342
>>> exp(-10000)
1.135483865314736098540939e-4343
```

Evaluation is supported for interval arguments via `mpmath.iv.exp()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.exp([-inf,0])
[0.0, 1.0]
>>> iv.exp([0,1])
[1.0, 2.718281828459045235360287495 58]
```

The exponential function can be evaluated efficiently to arbitrary precision:

```
>>> mp.dps = 10000
>>> exp(pi)
23.140692632779269005729...8984304016040616
```

Functional properties

Numerical verification of Euler's identity for the complex exponential function:

```
>>> mp.dps = 15
>>> exp(j*pi)+1
(0.0 + 1.22464679914735e-16j)
>>> chop(exp(j*pi)+1)
0.0
```

This recovers the coefficients (reciprocal factorials) in the Maclaurin series expansion of exp:

```
>>> nprint(taylor(exp, 0, 5))
[1.0, 1.0, 0.5, 0.166667, 0.0416667, 0.00833333]
```

The exponential function is its own derivative and antiderivative:

```
>>> exp(pi)
23.1406926327793
>>> diff(exp, pi)
23.1406926327793
>>> quad(exp, [-inf, pi])
23.1406926327793
```

The exponential function can be evaluated using various methods, including direct summation of the series, limits, and solving the defining differential equation:

```
>>> nsum(lambda k: pi**k/fac(k), [0,inf])
23.1406926327793
>>> limit(lambda k: (1+pi/k)**k, inf)
23.1406926327793
>>> odefun(lambda t, x: x, 0, 1)(pi)
23.1406926327793
```

power()

mpmath.**power**(x, y)

Converts x and y to mpmath numbers and evaluates $x^y = \exp(y \log(x))$:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> power(2, 0.5)
1.41421356237309504880168872421
```

This shows the leading few digits of a large Mersenne prime (performing the exact calculation `2**43112609-1` and displaying the result in Python would be very slow):

```
>>> power(2, 43112609)-1
3.16470269330255923143453723949e+12978188
```

expj()

mpmath.**expj**(x, ***kwargs*)

Convenience function for computing e^{ix}:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> expj(0)
(1.0 + 0.0j)
>>> expj(-1)
(0.5403023058681397174009366 - 0.8414709848078965066525023j)
>>> expj(j)
```

```
(0.367879441171442321595523 8 + 0.0j)
>>> expj(1+j)
(0.198766110346412940628803 2 + 0.309559875653112198443912 8j)
```

expjpi()

mpmath.**expjpi**(*x*, ***kwargs*)

Convenience function for computing $e^{i\pi x}$. Evaluation is accurate near zeros (see also `cospi()` (page 676), `sinpi()` (page 676)):

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> expjpi(0)
(1.0 + 0.0j)
>>> expjpi(1)
(-1.0 + 0.0j)
>>> expjpi(0.5)
(0.0 + 1.0j)
>>> expjpi(-1)
(-1.0 + 0.0j)
>>> expjpi(j)
(0.0432139182637722497744177 4 + 0.0j)
>>> expjpi(1+j)
(-0.0432139182637722497744177 4 + 0.0j)
```

expm1()

mpmath.**expm1**(*x*)

Computes $e^x - 1$, accurately for small x.

Unlike the expression `exp(x) - 1`, `expm1(x)` does not suffer from potentially catastrophic cancellation:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> exp(1e-10)-1; print(expm1(1e-10))
1.00000008274037e-10
1.00000000005e-10
>>> exp(1e-20)-1; print(expm1(1e-20))
0.0
1.0e-20
>>> 1/(exp(1e-20)-1)
Traceback (most recent call last):
  ...
ZeroDivisionError
>>> 1/expm1(1e-20)
1.0e+20
```

Evaluation works for extremely tiny values:

```
>>> expm1(0)
0.0
>>> expm1('1e-10000000')
1.0e-10000000
```

powm1()

mpmath.**powm1**(*x*, *y*)

Computes $x^y - 1$, accurately when x^y is very close to 1.

This avoids potentially catastrophic cancellation:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> power(0.99999995, 1e-10) - 1
0.0
>>> powm1(0.99999995, 1e-10)
-5.00000012791934e-18
```

Powers exactly equal to 1, and only those powers, yield 0 exactly:

```
>>> powm1(-j, 4)
(0.0 + 0.0j)
>>> powm1(3, 0)
0.0
>>> powm1(fadd(-1, 1e-100, exact=True), 4)
-4.0e-100
```

Evaluation works for extremely tiny y:

```
>>> powm1(2, '1e-100000')
6.93147180559945e-100001
>>> powm1(j, '1e-1000')
(-1.23370055013617e-2000 + 1.5707963267949e-1000j)
```

Logarithms

log()

mpmath.**log**(*x*, *b=None*)

Computes the base-b logarithm of x, $\log_b(x)$. If b is unspecified, `log()` (page 665) computes the natural (base e) logarithm and is equivalent to `ln()` (page 666). In general, the base b logarithm is defined in terms of the natural logarithm as $\log_b(x) = \ln(x)/\ln(b)$.

By convention, we take $\log(0) = -\infty$.

The natural logarithm is real if $x > 0$ and complex if $x < 0$ or if x is complex. The principal branch of the complex logarithm is used, meaning that $\Im(\ln(x)) = -\pi < \arg(x) \le \pi$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> log(1)
0.0
>>> log(2)
0.693147180559945
>>> log(1000,10)
3.0
>>> log(4, 16)
0.5
>>> log(j)
(0.0 + 1.5707963267949j)
>>> log(-1)
(0.0 + 3.14159265358979j)
```

```
>>> log(0)
-inf
>>> log(inf)
+inf
```

The natural logarithm is the antiderivative of $1/x$:

```
>>> quad(lambda x: 1/x, [1, 5])
1.6094379124341
>>> log(5)
1.6094379124341
>>> diff(log, 10)
0.1
```

The Taylor series expansion of the natural logarithm around $x = 1$ has coefficients $(-1)^{n+1}/n$:

```
>>> nprint(taylor(log, 1, 7))
[0.0, 1.0, -0.5, 0.333333, -0.25, 0.2, -0.166667, 0.142857]
```

`log()` (page 665) supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> log(pi)
1.1447298858494001741434273513530587116472948129153
>>> log(pi, pi**3)
0.33333333333333333333333333333333333333333333333333
>>> mp.dps = 25
>>> log(3+4j)
(1.6094379124341003746007591 + 0.92729521800161223242851255j)
```

ln()

mpmath.**ln**(*x*, ***kwargs*)

Computes the base-b logarithm of x, $\log_b(x)$. If b is unspecified, `log()` (page 665) computes the natural (base e) logarithm and is equivalent to `ln()` (page 666). In general, the base b logarithm is defined in terms of the natural logarithm as $\log_b(x) = \ln(x)/\ln(b)$.

By convention, we take $\log(0) = -\infty$.

The natural logarithm is real if $x > 0$ and complex if $x < 0$ or if x is complex. The principal branch of the complex logarithm is used, meaning that $\Im(\ln(x)) = -\pi < \arg(x) \le \pi$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> log(1)
0.0
>>> log(2)
0.693147180559945
>>> log(1000,10)
3.0
>>> log(4, 16)
0.5
>>> log(j)
(0.0 + 1.5707963267949j)
>>> log(-1)
```

```
(0.0 + 3.14159265358979j)
>>> log(0)
-inf
>>> log(inf)
+inf
```

The natural logarithm is the antiderivative of $1/x$:

```
>>> quad(lambda x: 1/x, [1, 5])
1.6094379124341
>>> log(5)
1.6094379124341
>>> diff(log, 10)
0.1
```

The Taylor series expansion of the natural logarithm around $x = 1$ has coefficients $(-1)^{n+1}/n$:

```
>>> nprint(taylor(log, 1, 7))
[0.0, 1.0, -0.5, 0.333333, -0.25, 0.2, -0.166667, 0.142857]
```

`log()` (page 665) supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> log(pi)
1.1447298858494001741434273513530587116472948129153
>>> log(pi, pi**3)
0.33333333333333333333333333333333333333333333333333
>>> mp.dps = 25
>>> log(3+4j)
(1.6094379124341003746007 59 + 0.92729521800161223242851 25j)
```

log10()

mpmath.**log10**(*x*)

Computes the base-10 logarithm of x, $\log_{10}(x)$. `log10(x)` is equivalent to `log(x, 10)`.

Lambert W function

lambertw()

mpmath.**lambertw**(*z*, *k=0*)

The Lambert W function $W(z)$ is defined as the inverse function of $w\exp(w)$. 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 $z = w\exp(w)$. All branches are supported by `lambertw()` (page 667):

- `lambertw(z)` gives the principal solution (branch 0)
- `lambertw(z, k)` gives the solution on branch k

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 $z = 0$.

The definition, implementation and choice of branches is based on [Corless] (page 1625).

Plots

```
# Branches 0 and -1 of the Lambert W function
plot([lambertw, lambda x: lambertw(x,-1)], [-2,2], [-5,2], points=2000)
```

```
# Principal branch of the Lambert W function W(z)
cplot(lambertw, [-1,1], [-1,1], points=50000)
```

Basic examples

The Lambert W function is the inverse of $w \exp(w)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> w = lambertw(1)
>>> w
0.5671432904097838729999687
>>> w*exp(w)
1.0
```

Any branch gives a valid inverse:

```
>>> w = lambertw(1, k=3)
>>> w
(-2.853581755409037807206819 + 17.11353553941214591260783j)
>>> w = lambertw(1, k=25)
>>> w
(-5.047020464221569709378686 + 155.4763860949415867162066j)
>>> chop(w*exp(w))
1.0
```

Applications to equation-solving

The Lambert W function may be used to solve various kinds of equations, such as finding the value of the infinite power tower $z^{z^{z^{\cdots}}}$:

```
>>> def tower(z, n):
...     if n == 0:
...         return z
...     return z ** tower(z, n-1)
...
>>> tower(mpf(0.5), 100)
0.641185744504985984486200 5
>>> -lambertw(-log(0.5))/log(0.5)
0.641185744504985984486200 5
```

Properties

The Lambert W function grows roughly like the natural logarithm for large arguments:

```
>>> lambertw(1000); log(1000)
5.2496028524015962271260 56
6.9077552789821370520539 74
>>> lambertw(10**100); log(10**100)
224.8431064451185015393731
230.2585092994045684017991
```

The principal branch of the Lambert W function has a rational Taylor series expansion around $z = 0$:

```
>>> nprint(taylor(lambertw, 0, 6), 10)
[0.0, 1.0, -1.0, 1.5, -2.666666667, 5.208333333, -10.8]
```

Some special values and limits are:

```
>>> lambertw(0)
0.0
>>> lambertw(1)
0.567143290409783872999968 7
>>> lambertw(e)
1.0
>>> lambertw(inf)
+inf
>>> lambertw(0, k=-1)
-inf
>>> lambertw(0, k=3)
-inf
>>> lambertw(inf, k=2)
(+inf + 12.56637061435917295385057j)
>>> lambertw(inf, k=3)
(+inf + 18.84955592153875943077586j)
>>> lambertw(-inf, k=3)
(+inf + 21.99114857512855266923 85j)
```

The $k = 0$ and $k = -1$ branches join at $z = -1/e$ where $W(z) = -1$ for both branches. Since $-1/e$ can only be represented approximately with binary floating-point numbers, evaluating the Lambert W function at this point only gives -1 approximately:

```
>>> lambertw(-1/e, 0)
-0.999999999999983713302282 51
>>> lambertw(-1/e, -1)
-1.000000000000016286697717 5
```

If $-1/e$ happens to round in the negative direction, there might be a small imaginary part:

```
>>> mp.dps = 15
>>> lambertw(-1/e)
(-1.0 + 8.22007971483662e-9j)
>>> lambertw(-1/e+eps)
-0.999999966242188
```

References

1. [Corless] (page 1625)

Arithmetic-geometric mean

agm()

mpmath.**agm**(*a*, *b=1*)

`agm(a, b)` computes the arithmetic-geometric mean of a and b, defined as the limit of the following iteration:

$$a_0 = a$$
$$b_0 = b$$
$$a_{n+1} = \frac{a_n + b_n}{2}$$
$$b_{n+1} = \sqrt{a_n b_n}$$

This function can be called with a single argument, computing $\mathrm{agm}(a,1) = \mathrm{agm}(1,a)$.

Examples

It is a well-known theorem that the geometric mean of two distinct positive numbers is less than the arithmetic mean. It follows that the arithmetic-geometric mean lies between the two means:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> a = mpf(3)
>>> b = mpf(4)
>>> sqrt(a*b)
3.46410161513775
>>> agm(a,b)
3.48202767635957
>>> (a+b)/2
3.5
```

The arithmetic-geometric mean is scale-invariant:

```
>>> agm(10*e, 10*pi)
29.261085515723
>>> 10*agm(e, pi)
29.261085515723
```

As an order-of-magnitude estimate, $\mathrm{agm}(1,x) \approx x$ for large x:

```
>>> agm(10**10)
643448704.760133
>>> agm(10**50)
1.34814309345871e+48
```

For tiny x, $\mathrm{agm}(1,x) \approx -\pi/(2\log(x/4))$:

```
>>> agm('0.01')
0.262166887202249
>>> -pi/2/log('0.0025')
0.262172347753122
```

The arithmetic-geometric mean can also be computed for complex numbers:

```
>>> agm(3, 2+j)
(2.51055133276184 + 0.547394054060638j)
```

The AGM iteration converges very quickly (each step doubles the number of correct digits), so `agm()` (page 671) supports efficient high-precision evaluation:

```
>>> mp.dps = 10000
>>> a = agm(1,2)
>>> str(a)[-10:]
'1679581912'
```

Mathematical relations

The arithmetic-geometric mean may be used to evaluate the following two parametric definite integrals:

$$I_1 = \int_0^\infty \frac{1}{\sqrt{(x^2+a^2)(x^2+b^2)}}\,dx$$

$$I_2 = \int_0^{\pi/2} \frac{1}{\sqrt{a^2\cos^2(x)+b^2\sin^2(x)}}\,dx$$

We have:

```
>>> mp.dps = 15
>>> a = 3
>>> b = 4
>>> f1 = lambda x: ((x**2+a**2)*(x**2+b**2))**-0.5
>>> f2 = lambda x: ((a*cos(x))**2 + (b*sin(x))**2)**-0.5
>>> quad(f1, [0, inf])
0.451115405388492
>>> quad(f2, [0, pi/2])
0.451115405388492
>>> pi/(2*agm(a,b))
0.451115405388492
```

A formula for $\Gamma(1/4)$:

```
>>> gamma(0.25)
3.62560990822191
>>> sqrt(2*sqrt(2*pi**3)/agm(1,sqrt(2)))
3.62560990822191
```

Possible issues

The branch cut chosen for complex a and b is somewhat arbitrary.

Trigonometric functions

Except where otherwise noted, the trigonometric functions take a radian angle as input and the inverse trigonometric functions return radian angles.

The ordinary trigonometric functions are single-valued functions defined everywhere in the complex plane (except at the poles of tan, sec, csc, and cot). They are defined generally via the exponential function, e.g.

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}.$$

The inverse trigonometric functions are multivalued, thus requiring branch cuts, and are generally real-valued only on a part of the real line. Definitions and branch cuts are given in the documentation of each function. The branch cut conventions used by mpmath are essentially the same as those found in most standard mathematical software, such as Mathematica and Python's own `cmath` libary (as of Python 2.6; earlier Python versions implement some functions erroneously).

Degree-radian conversion

degrees()

mpmath.**degrees**(*x*)

Converts the radian angle x to a degree angle:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> degrees(pi/3)
60.0
```

radians()

mpmath.**radians**(*x*)

Converts the degree angle x to radians:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> radians(60)
1.0471975511966
```

Trigonometric functions

cos()

mpmath.**cos**(*x*, ***kwargs*)

Computes the cosine of x, $\cos(x)$.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> cos(pi/3)
0.5
>>> cos(100000001)
-0.9802850113244713353133243
>>> cos(2+3j)
(-4.189625690968807230132555 - 9.109227893755336597979197j)
>>> cos(inf)
nan
>>> nprint(chop(taylor(cos, 0, 6)))
[1.0, 0.0, -0.5, 0.0, 0.0416667, 0.0, -0.00138889]
```

Intervals are supported via `mpmath.iv.cos()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.cos([0,1])
[0.5403023058681397174009366023 01, 1.0]
>>> iv.cos([0,2])
[-0.4161468365471423869975682321 4, 1.0]
```

sin()

mpmath.**sin**(*x*, ***kwargs*)

Computes the sine of x, $\sin(x)$.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sin(pi/3)
0.866025403784438646763723 2
>>> sin(100000001)
0.197588705579496891143874 3
>>> sin(2+3j)
(9.154499146911429573467 3 - 4.168906959966564350754813j)
>>> sin(inf)
nan
>>> nprint(chop(taylor(sin, 0, 6)))
[0.0, 1.0, 0.0, -0.166667, 0.0, 0.00833333, 0.0]
```

Intervals are supported via `mpmath.iv.sin()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.sin([0,1])
[0.0, 0.8414709848078965066525023312 01]
>>> iv.sin([0,2])
[0.0, 1.0]
```

tan()

mpmath.**tan**(*x*, ***kwargs*)

Computes the tangent of x, $\tan(x) = \frac{\sin(x)}{\cos(x)}$. The tangent function is singular at $x = (n + 1/2)\pi$, but `tan(x)` always returns a finite result since $(n + 1/2)\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> tan(pi/3)
1.732050807568877293527446
>>> tan(100000001)
-0.2015625081449864533091058
>>> tan(2+3j)
(-0.00376402564150424829275122 1 + 1.00323862735360980144635 9j)
>>> tan(inf)
nan
>>> nprint(chop(taylor(tan, 0, 6)))
[0.0, 1.0, 0.0, 0.333333, 0.0, 0.133333, 0.0]
```

Intervals are supported via `mpmath.iv.tan()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.tan([0,1])
```

```
[0.0, 1.5574077246549022305069748294 4]
>>> iv.tan([0,2])  # Interval includes a singularity
[-inf, +inf]
```

sec()

mpmath.**sec**(x)

Computes the secant of x, $\sec(x) = \frac{1}{\cos(x)}$. The secant function is singular at $x = (n+1/2)\pi$, but `sec(x)` always returns a finite result since $(n + 1/2)\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sec(pi/3)
2.0
>>> sec(10000001)
-1.184723164360392819100265
>>> sec(2+3j)
(-0.041674964411144270048349 91 + 0.090611137196237596529661 2j)
>>> sec(inf)
nan
>>> nprint(chop(taylor(sec, 0, 6)))
[1.0, 0.0, 0.5, 0.0, 0.208333, 0.0, 0.0847222]
```

Intervals are supported via `mpmath.iv.sec()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.sec([0,1])
[1.0, 1.850815717680925617911753262 76]
>>> iv.sec([0,2])  # Interval includes a singularity
[-inf, +inf]
```

csc()

mpmath.**csc**(x)

Computes the cosecant of x, $\csc(x) = \frac{1}{\sin(x)}$. This cosecant function is singular at $x = n\pi$, but with the exception of the point $x = 0$, `csc(x)` returns a finite result since $n\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> csc(pi/3)
1.154700538379251529018298
>>> csc(10000001)
-1.864910497503629858938891
>>> csc(2+3j)
(0.090473209753207439805790 48 + 0.041200986288574126463009 81j)
>>> csc(inf)
nan
```

Intervals are supported via `mpmath.iv.csc()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.csc([0,1])  # Interval includes a singularity
[1.188395105778121216261599439 88, +inf]
>>> iv.csc([0,2])
[1.0, +inf]
```

cot()

mpmath.**cot**(*x*)

Computes the cotangent of x, $\cot(x) = \frac{1}{\tan(x)} = \frac{\cos(x)}{\sin(x)}$. This cotangent function is singular at $x = n\pi$, but with the exception of the point $x = 0$, `cot(x)` returns a finite result since $n\pi$ cannot be represented exactly using floating-point arithmetic.

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> cot(pi/3)
0.5773502691896257645091488
>>> cot(10000001)
1.574131876209625656003562
>>> cot(2+3j)
(-0.003739710376336956660117409 - 0.9967577965693583104609688j)
>>> cot(inf)
nan
```

Intervals are supported via `mpmath.iv.cot()`:

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.cot([0,1])  # Interval includes a singularity
[0.6420926159343307030064199748 62, +inf]
>>> iv.cot([1,2])
[-inf, +inf]
```

Trigonometric functions with modified argument

cospi()

mpmath.**cospi**(*x*, ***kwargs*)

Computes $\cos(\pi x)$, more accurately than the expression `cos(pi*x)`:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> cospi(10**10), cos(pi*(10**10))
(1.0, 0.999999999997493)
>>> cospi(10**10+0.5), cos(pi*(10**10+0.5))
(0.0, 1.59960492420134e-6)
```

sinpi()

mpmath.**sinpi**(*x*, ***kwargs*)

Computes $\sin(\pi x)$, more accurately than the expression `sin(pi*x)`:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sinpi(10**10), sin(pi*(10**10))
(0.0, -2.23936276195592e-6)
>>> sinpi(10**10+0.5), sin(pi*(10**10+0.5))
(1.0, 0.999999999998721)
```

Inverse trigonometric functions

acos()

mpmath.**acos**(*x*, ***kwargs*)

Computes the inverse cosine or arccosine of x, $\cos^{-1}(x)$. Since $-1 \le \cos(x) \le 1$ for real x, the inverse cosine is real-valued only for $-1 \le x \le 1$. On this interval, acos() (page 677) is defined to be a monotonically decreasing function assuming values between $+\pi$ and 0.

Basic values are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> acos(-1)
3.141592653589793238462643
>>> acos(0)
1.570796326794896619231322
>>> acos(1)
0.0
>>> nprint(chop(taylor(acos, 0, 6)))
[1.5708, -1.0, 0.0, -0.166667, 0.0, -0.075, 0.0]
```

acos() (page 677) is defined so as to be a proper inverse function of $\cos(\theta)$ for $0 \le \theta < \pi$. We have $\cos(\cos^{-1}(x)) = x$ for all x, but $\cos^{-1}(\cos(x)) = x$ only for $0 \le \Re[x] < \pi$:

```
>>> for x in [1, 10, -1, 2+3j, 10+3j]:
...     print("%s %s" % (cos(acos(x)), acos(cos(x))))
...
1.0 1.0
(10.0 + 0.0j) 2.566370614359172953850574
-1.0 1.0
(2.0 + 3.0j) (2.0 + 3.0j)
(10.0 + 3.0j) (2.566370614359172953850574 - 3.0j)
```

The inverse cosine has two branch points: $x = \pm 1$. acos() (page 677) places the branch cuts along the line segments $(-\infty, -1)$ and $(+1, +\infty)$. In general,

$$\cos^{-1}(x) = \frac{\pi}{2} + i \log\left(ix + \sqrt{1 - x^2}\right)$$

where the principal-branch log and square root are implied.

asin()

mpmath.**asin**(*x*, ***kwargs*)

Computes the inverse sine or arcsine of x, $\sin^{-1}(x)$. Since $-1 \le \sin(x) \le 1$ for real x, the inverse sine is real-valued only for $-1 \le x \le 1$. On this interval, it is defined to be a monotonically increasing function assuming values between $-\pi/2$ and $\pi/2$.

Basic values are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> asin(-1)
-1.570796326794896619231322
>>> asin(0)
0.0
>>> asin(1)
1.570796326794896619231322
>>> nprint(chop(taylor(asin, 0, 6)))
[0.0, 1.0, 0.0, 0.166667, 0.0, 0.075, 0.0]
```

asin() (page 677) is defined so as to be a proper inverse function of $\sin(\theta)$ for $-\pi/2 < \theta < \pi/2$. We have $\sin(\sin^{-1}(x)) = x$ for all x, but $\sin^{-1}(\sin(x)) = x$ only for $-\pi/2 < \Re[x] < \pi/2$:

```
>>> for x in [1, 10, -1, 1+3j, -2+3j]:
...     print("%s %s" % (chop(sin(asin(x))), asin(sin(x))))
...
1.0 1.0
10.0 -0.575222039230620284612069 8
-1.0 -1.0
(1.0 + 3.0j) (1.0 + 3.0j)
(-2.0 + 3.0j) (-1.14159265358979323846264 3 - 3.0j)
```

The inverse sine has two branch points: $x = \pm 1$. asin() (page 677) places the branch cuts along the line segments $(-\infty, -1)$ and $(+1, +\infty)$. In general,

$$\sin^{-1}(x) = -i \log\left(ix + \sqrt{1 - x^2}\right)$$

where the principal-branch log and square root are implied.

atan()

mpmath.**atan**(*x*, ***kwargs*)

Computes the inverse tangent or arctangent of x, $\tan^{-1}(x)$. This is a real-valued function for all real x, with range $(-\pi/2, \pi/2)$.

Basic values are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> atan(-inf)
-1.570796326794896619231322
>>> atan(-1)
-0.7853981633974483096156609
>>> atan(0)
0.0
>>> atan(1)
0.7853981633974483096156609
>>> atan(inf)
1.570796326794896619231322
>>> nprint(chop(taylor(atan, 0, 6)))
[0.0, 1.0, 0.0, -0.333333, 0.0, 0.2, 0.0]
```

The inverse tangent is often used to compute angles. However, the atan2 function is often better for this as it preserves sign (see atan2() (page 679)).

atan() (page 678) is defined so as to be a proper inverse function of $\tan(\theta)$ for $-\pi/2 < \theta < \pi/2$. We have $\tan(\tan^{-1}(x)) = x$ for all x, but $\tan^{-1}(\tan(x)) = x$ only for $-\pi/2 < \Re[x] < \pi/2$:

```
>>> mp.dps = 25
>>> for x in [1, 10, -1, 1+3j, -2+3j]:
...     print("%s %s" % (tan(atan(x)), atan(tan(x))))
...
1.0 1.0
10.0 0.5752220392306202846120698
-1.0 -1.0
(1.0 + 3.0j) (1.000000000000000000000001 + 3.0j)
(-2.0 + 3.0j) (1.14159265358979323846264 4 + 3.0j)
```

The inverse tangent has two branch points: $x = \pm i$. `atan()` (page 678) places the branch cuts along the line segments $(-i\infty, -i)$ and $(+i, +i\infty)$. In general,

$$\tan^{-1}(x) = \frac{i}{2}\left(\log(1 - ix) - \log(1 + ix)\right)$$

where the principal-branch log is implied.

atan2()

mpmath.**atan2**(*y*, *x*)

Computes the two-argument arctangent, $\operatorname{atan2}(y, x)$, giving the signed angle between the positive x-axis and the point (x, y) in the 2D plane. This function is defined for real x and y only.

The two-argument arctangent essentially computes $\operatorname{atan}(y/x)$, but accounts for the signs of both x and y to give the angle for the correct quadrant. The following examples illustrate the difference:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> atan2(1,1), atan(1/1.)
(0.785398163397448, 0.785398163397448)
>>> atan2(1,-1), atan(1/-1.)
(2.35619449019234, -0.785398163397448)
>>> atan2(-1,1), atan(-1/1.)
(-0.785398163397448, -0.785398163397448)
>>> atan2(-1,-1), atan(-1/-1.)
(-2.35619449019234, 0.785398163397448)
```

The angle convention is the same as that used for the complex argument; see `arg()` (page 641).

asec()

mpmath.**asec**(*x*)

Computes the inverse secant of x, $\sec^{-1}(x) = \cos^{-1}(1/x)$.

acsc()

mpmath.**acsc**(*x*)

Computes the inverse cosecant of x, $\csc^{-1}(x) = \sin^{-1}(1/x)$.

acot()

mpmath.**acot**(*x*)

Computes the inverse cotangent of x, $\cot^{-1}(x) = \tan^{-1}(1/x)$.

Sinc function

sinc()

mpmath.**sinc**(*x*)

`sinc(x)` computes the unnormalized sinc function, defined as

$$\operatorname{sinc}(x) = \begin{cases} \sin(x)/x, & \text{if } x \neq 0 \\ 1, & \text{if } x = 0. \end{cases}$$

See sincpi() (page 680) for the normalized sinc function.

Simple values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sinc(0)
1.0
>>> sinc(1)
0.841470984807897
>>> sinc(inf)
0.0
```

The integral of the sinc function is the sine integral Si:

```
>>> quad(sinc, [0, 1])
0.946083070367183
>>> si(1)
0.946083070367183
```

sincpi()

mpmath.**sincpi**(x)

sincpi(x) computes the normalized sinc function, defined as

$$\operatorname{sinc}_\pi(x) = \begin{cases} \sin(\pi x)/(\pi x), & \text{if } x \neq 0 \\ 1, & \text{if } x = 0. \end{cases}$$

Equivalently, we have $\operatorname{sinc}_\pi(x) = \operatorname{sinc}(\pi x)$.

The normalization entails that the function integrates to unity over the entire real line:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> quadosc(sincpi, [-inf, inf], period=2.0)
1.0
```

Like, sinpi() (page 676), sincpi() (page 680) is evaluated accurately at its roots:

```
>>> sincpi(10)
0.0
```

Hyperbolic functions

Hyperbolic functions

cosh()

mpmath.**cosh**(x, ***kwargs*)

Computes the hyperbolic cosine of x, $\cosh(x) = (e^x + e^{-x})/2$. Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> cosh(0)
1.0
>>> cosh(1)
1.543080634815243778477906
>>> cosh(-inf), cosh(+inf)
(+inf, +inf)
```

The hyperbolic cosine is an even, convex function with a global minimum at $x = 0$, having a Maclaurin series that starts:

```
>>> nprint(chop(taylor(cosh, 0, 5)))
[1.0, 0.0, 0.5, 0.0, 0.0416667, 0.0]
```

Generalized to complex numbers, the hyperbolic cosine is equivalent to a cosine with the argument rotated in the imaginary direction, or $\cosh x = \cos ix$:

```
>>> cosh(2+3j)
(-3.724545504915322565473971 + 0.5118225699873846088344638j)
>>> cos(3-2j)
(-3.724545504915322565473971 + 0.5118225699873846088344638j)
```

sinh()

mpmath.**sinh**(*x*, ***kwargs*)

Computes the hyperbolic sine of x, $\sinh(x) = (e^x - e^{-x})/2$. Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sinh(0)
0.0
>>> sinh(1)
1.175201193643801456882382
>>> sinh(-inf), sinh(+inf)
(-inf, +inf)
```

The hyperbolic sine is an odd function, with a Maclaurin series that starts:

```
>>> nprint(chop(taylor(sinh, 0, 5)))
[0.0, 1.0, 0.0, 0.166667, 0.0, 0.00833333]
```

Generalized to complex numbers, the hyperbolic sine is essentially a sine with a rotation i applied to the argument; more precisely, $\sinh x = -i \sin ix$:

```
>>> sinh(2+3j)
(-3.590564589985779952012565 + 0.5309210862485198052670401j)
>>> j*sin(3-2j)
(-3.590564589985779952012565 + 0.5309210862485198052670401j)
```

tanh()

mpmath.**tanh**(*x*, ***kwargs*)

Computes the hyperbolic tangent of x, $\tanh(x) = \sinh(x)/\cosh(x)$. Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> tanh(0)
0.0
>>> tanh(1)
0.7615941559557648881194583
>>> tanh(-inf), tanh(inf)
(-1.0, 1.0)
```

The hyperbolic tangent is an odd, sigmoidal function, similar to the inverse tangent and error function. Its Maclaurin series is:

```
>>> nprint(chop(taylor(tanh, 0, 5)))
[0.0, 1.0, 0.0, -0.333333, 0.0, 0.133333]
```

Generalized to complex numbers, the hyperbolic tangent is essentially a tangent with a rotation i applied to the argument; more precisely, $\tanh x = -i \tan ix$:

```
>>> tanh(2+3j)
(0.965385879022133124278480 3 - 0.00988437503832249372031403 4j)
>>> j*tan(3-2j)
(0.965385879022133124278480 3 - 0.00988437503832249372031403 4j)
```

sech()
mpmath.**sech**(*x*)
Computes the hyperbolic secant of x, $\operatorname{sech}(x) = \frac{1}{\cosh(x)}$.

csch()
mpmath.**csch**(*x*)
Computes the hyperbolic cosecant of x, $\operatorname{csch}(x) = \frac{1}{\sinh(x)}$.

coth()
mpmath.**coth**(*x*)
Computes the hyperbolic cotangent of x, $\coth(x) = \frac{\cosh(x)}{\sinh(x)}$.

Inverse hyperbolic functions

acosh()
mpmath.**acosh**(*x*, ***kwargs*)
Computes the inverse hyperbolic cosine of x, $\cosh^{-1}(x) = \log(x + \sqrt{x+1}\sqrt{x-1})$.

asinh()
mpmath.**asinh**(*x*, ***kwargs*)
Computes the inverse hyperbolic sine of x, $\sinh^{-1}(x) = \log(x + \sqrt{1+x^2})$.

atanh()
mpmath.**atanh**(*x*, ***kwargs*)
Computes the inverse hyperbolic tangent of x, $\tanh^{-1}(x) = \frac{1}{2}\left(\log(1+x) - \log(1-x)\right)$.

asech()
mpmath.**asech**(*x*)
Computes the inverse hyperbolic secant of x, $\operatorname{sech}^{-1}(x) = \cosh^{-1}(1/x)$.

acsch()
mpmath.**acsch**(*x*)
Computes the inverse hyperbolic cosecant of x, $\operatorname{csch}^{-1}(x) = \sinh^{-1}(1/x)$.

acoth()
mpmath.**acoth**(*x*)
Computes the inverse hyperbolic cotangent of x, $\coth^{-1}(x) = \tanh^{-1}(1/x)$.

Factorials and gamma functions

Factorials and factorial-like sums and products are basic tools of combinatorics and number theory. Much like the exponential function is fundamental to differential equations and analysis in general, the factorial function (and its extension to complex numbers, the gamma function) is fundamental to difference equations and functional equations.

A large selection of factorial-like functions is implemented in mpmath. All functions support complex arguments, and arguments may be arbitrarily large. Results are numerical approximations, so to compute *exact* values a high enough precision must be set manually:

```
>>> mp.dps = 15; mp.pretty = True
>>> fac(100)
9.33262154439442e+157
>>> print int(_)    # most digits are wrong
93326215443944150965646704795953882578400970373184098831012889540582227238570431
29506611308928832727782584966400652427055453597628971938285218186589595972403 2
>>> mp.dps = 160
>>> fac(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915
608941463976156518286253697920827223758251185210916864000000000000000000000000.0
```

The gamma and polygamma functions are closely related to *Zeta functions, L-series and polylogarithms* (page 836). See also *q-functions* (page 871) for q-analogs of factorial-like functions.

Factorials

factorial()/fac()
mpmath.**factorial**(*x*, ***kwargs*)
Computes the factorial, $x!$. For integers $n \geq 0$, we have $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ and more generally the factorial is defined for real or complex x by $x! = \Gamma(x+1)$.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(6):
...     print("%s %s" % (k, fac(k)))
...
0 1.0
1 1.0
2 2.0
3 6.0
4 24.0
5 120.0
>>> fac(inf)
+inf
>>> fac(0.5), sqrt(pi)/2
(0.886226925452758, 0.886226925452758)
```

For large positive x, $x!$ can be approximated by Stirling's formula:

```
>>> x = 10**10
>>> fac(x)
2.32579620567308e+95657055186
>>> sqrt(2*pi*x)*(x/e)**x
2.32579597597705e+95657055186
```

`fac()` supports evaluation for astronomically large values:

```
>>> fac(10**30)
6.22311232304258e+295657055180967481723488710810 98
```

Reciprocal factorials appear in the Taylor series of the exponential function (among many other contexts):

```
>>> nsum(lambda k: 1/fac(k), [0, inf]), exp(1)
(2.71828182845905, 2.71828182845905)
>>> nsum(lambda k: pi**k/fac(k), [0, inf]), exp(pi)
(23.1406926327793, 23.1406926327793)
```

fac2()

mpmath.**fac2**(x)

Computes the double factorial $x!!$, defined for integers $x > 0$ by

$$x!! = \begin{cases} 1 \cdot 3 \cdots (x-2) \cdot x & x \text{ odd} \\ 2 \cdot 4 \cdots (x-2) \cdot x & x \text{ even} \end{cases}$$

and more generally by [1]

$$x!! = 2^{x/2} \left(\frac{\pi}{2}\right)^{(\cos(\pi x)-1)/4} \Gamma\left(\frac{x}{2}+1\right).$$

Examples

The integer sequence of double factorials begins:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint([fac2(n) for n in range(10)])
[1.0, 1.0, 2.0, 3.0, 8.0, 15.0, 48.0, 105.0, 384.0, 945.0]
```

For large x, double factorials follow a Stirling-like asymptotic approximation:

```
>>> x = mpf(10000)
>>> fac2(x)
5.97272691416282e+17830
>>> sqrt(pi)*x**((x+1)/2)*exp(-x/2)
5.97262736954392e+17830
```

The recurrence formula $x!! = x(x-2)!!$ can be reversed to define the double factorial of negative odd integers (but not negative even integers):

```
>>> fac2(-1), fac2(-3), fac2(-5), fac2(-7)
(1.0, -1.0, 0.333333333333333, -0.0666666666666667)
>>> fac2(-2)
Traceback (most recent call last):
  ...
ValueError: gamma function pole
```

With the exception of the poles at negative even integers, fac2() (page 684) supports evaluation for arbitrary complex arguments. The recurrence formula is valid generally:

```
>>> fac2(pi+2j)
(-1.3697207890154e-12 + 3.93665300979176e-12j)
>>> (pi+2j)*fac2(pi-2+2j)
(-1.3697207890154e-12 + 3.93665300979176e-12j)
```

Double factorials should not be confused with nested factorials, which are immensely larger:

```
>>> fac(fac(20))
5.13805976125208e+43675043585825292774
>>> fac2(20)
3715891200.0
```

Double factorials appear, among other things, in series expansions of Gaussian functions and the error function. Infinite series include:

```
>>> nsum(lambda k: 1/fac2(k), [0, inf])
3.05940740534258
>>> sqrt(e)*(1+sqrt(pi/2)*erf(sqrt(2)/2))
3.05940740534258
>>> nsum(lambda k: 2**k/fac2(2*k-1), [1, inf])
4.06015693855741
>>> e * erf(1) * sqrt(pi)
4.06015693855741
```

A beautiful Ramanujan sum:

```
>>> nsum(lambda k: (-1)**k*(fac2(2*k-1)/fac2(2*k))**3, [0,inf])
0.90917279454693
>>> (gamma('9/8')/gamma('5/4')/gamma('7/8'))**2
0.90917279454693
```

References

1. http://functions.wolfram.com/GammaBetaErf/Factorial2/27/01/0002/
2. http://mathworld.wolfram.com/DoubleFactorial.html

Binomial coefficients

binomial()

mpmath.**binomial**(*n*, *k*)

Computes the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

The binomial coefficient gives the number of ways that k items can be chosen from a set of n items. More generally, the binomial coefficient is a well-defined function of arbitrary real or complex n and k, via the gamma function.

Examples

Generate Pascal's triangle:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint([binomial(n,k) for k in range(n+1)])
...
[1.0]
[1.0, 1.0]
[1.0, 2.0, 1.0]
[1.0, 3.0, 3.0, 1.0]
[1.0, 4.0, 6.0, 4.0, 1.0]
```

There is 1 way to select 0 items from the empty set, and 0 ways to select 1 item from the empty set:

```
>>> binomial(0, 0)
1.0
>>> binomial(0, 1)
0.0
```

`binomial()` (page 685) supports large arguments:

```
>>> binomial(10**20, 10**20-5)
8.33333333333333e+97
>>> binomial(10**20, 10**10)
2.60784095465201e+104342944813
```

Nonintegral binomial coefficients find use in series expansions:

```
>>> nprint(taylor(lambda x: (1+x)**0.25, 0, 4))
[1.0, 0.25, -0.09375, 0.0546875, -0.0375977]
>>> nprint([binomial(0.25, k) for k in range(5)])
[1.0, 0.25, -0.09375, 0.0546875, -0.0375977]
```

An integral representation:

```
>>> n, k = 5, 3
>>> f = lambda t: exp(-j*k*t)*(1+exp(j*t))**n
>>> chop(quad(f, [-pi,pi])/(2*pi))
10.0
>>> binomial(n,k)
10.0
```

Gamma function

gamma()

mpmath.**gamma**(*x*, ***kwargs*)

Computes the gamma function, $\Gamma(x)$. The gamma function is a shifted version of the ordinary factorial, satisfying $\Gamma(n) = (n-1)!$ for integers $n > 0$. More generally, it is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt$$

for any real or complex x with $\Re(x) > 0$ and for $\Re(x) < 0$ by analytic continuation.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(1, 6):
...     print("%s %s" % (k, gamma(k)))
...
1 1.0
2 1.0
3 2.0
4 6.0
5 24.0
>>> gamma(inf)
+inf
>>> gamma(0)
Traceback (most recent call last):
  ...
ValueError: gamma function pole
```

The gamma function of a half-integer is a rational multiple of $\sqrt{\pi}$:

```
>>> gamma(0.5), sqrt(pi)
(1.77245385090552, 1.77245385090552)
>>> gamma(1.5), sqrt(pi)/2
(0.886226925452758, 0.886226925452758)
```

We can check the integral definition:

```
>>> gamma(3.5)
3.32335097044784
>>> quad(lambda t: t**2.5*exp(-t), [0,inf])
3.32335097044784
```

`gamma()` (page 686) supports arbitrary-precision evaluation and complex arguments:

```
>>> mp.dps = 50
>>> gamma(sqrt(3))
0.91510229697308632046045539308226554038315280564184
>>> mp.dps = 25
>>> gamma(2j)
(0.009902440080927490985955066 - 0.075952001335018068724080 48j)
```

Arguments can also be large. Note that the gamma function grows very quickly:

```
>>> mp.dps = 15
>>> gamma(10**20)
1.9328495143101e+195657055180967481722 5
```

rgamma()

mpmath.**rgamma**(*x*, ***kwargs*)

Computes the reciprocal of the gamma function, $1/\Gamma(z)$. This function evaluates to zero at the poles of the gamma function, $z = 0, -1, -2, \ldots$.

Examples

Basic examples:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> rgamma(1)
1.0
```

```
>>> rgamma(4)
0.166666666666666666666667
>>> rgamma(0); rgamma(-1)
0.0
0.0
>>> rgamma(1000)
2.485168143266784862783596e-2565
>>> rgamma(inf)
0.0
```

A definite integral that can be evaluated in terms of elementary integrals:

```
>>> quad(rgamma, [0,inf])
2.807770242028519365221501
>>> e + quad(lambda t: exp(-t)/(pi**2+log(t)**2), [0,inf])
2.807770242028519365221501
```

gammaprod()

mpmath.**gammaprod**(a, b)

Given iterables a and b, `gammaprod(a, b)` computes the product / quotient of gamma functions:

$$\frac{\Gamma(a_0)\Gamma(a_1)\cdots\Gamma(a_p)}{\Gamma(b_0)\Gamma(b_1)\cdots\Gamma(b_q)}$$

Unlike direct calls to `gamma()` (page 686), `gammaprod()` (page 688) considers the entire product as a limit and evaluates this limit properly if any of the numerator or denominator arguments are nonpositive integers such that poles of the gamma function are encountered. That is, `gammaprod()` (page 688) evaluates

$$\lim_{\epsilon\to 0}\frac{\Gamma(a_0+\epsilon)\Gamma(a_1+\epsilon)\cdots\Gamma(a_p+\epsilon)}{\Gamma(b_0+\epsilon)\Gamma(b_1+\epsilon)\cdots\Gamma(b_q+\epsilon)}$$

In particular:

- If there are equally many poles in the numerator and the denominator, the limit is a rational number times the remaining, regular part of the product.
- If there are more poles in the numerator, `gammaprod()` (page 688) returns `+inf`.
- If there are more poles in the denominator, `gammaprod()` (page 688) returns 0.

Examples

The reciprocal gamma function $1/\Gamma(x)$ evaluated at $x = 0$:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> gammaprod([], [0])
0.0
```

A limit:

```
>>> gammaprod([-4], [-3])
-0.25
>>> limit(lambda x: gamma(x-1)/gamma(x), -3, direction=1)
-0.25
>>> limit(lambda x: gamma(x-1)/gamma(x), -3, direction=-1)
-0.25
```

loggamma()

mpmath.**loggamma**(*x*)

Computes the principal branch of the log-gamma function, $\ln \Gamma(z)$. Unlike $\ln(\Gamma(z))$, which has infinitely many complex branch cuts, the principal log-gamma function only has a single branch cut along the negative half-axis. The principal branch continuously matches the asymptotic Stirling expansion

$$\ln \Gamma(z) \sim \frac{\ln(2\pi)}{2} + \left(z - \frac{1}{2}\right) \ln(z) - z + O(z^{-1}).$$

The real parts of both functions agree, but their imaginary parts generally differ by $2n\pi$ for some $n \in \mathbb{Z}$. They coincide for $z \in \mathbb{R}, z > 0$.

Computationally, it is advantageous to use loggamma() (page 689) instead of gamma() (page 686) for extremely large arguments.

Examples

Comparing with $\ln(\Gamma(z))$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> loggamma('13.2'); log(gamma('13.2'))
20.49400419456603678498394
20.49400419456603678498394
>>> loggamma(3+4j)
(-1.756626784603784110530604 + 4.742664438034657928194889j)
>>> log(gamma(3+4j))
(-1.756626784603784110530604 - 1.540520869144928548730397j)
>>> log(gamma(3+4j)) + 2*pi*j
(-1.756626784603784110530604 + 4.742664438034657928194889j)
```

Note the imaginary parts for negative arguments:

```
>>> loggamma(-0.5); loggamma(-1.5); loggamma(-2.5)
(1.265512123484645396488946 - 3.141592653589793238462643j)
(0.860047015376481014510932 7 - 6.283185307179586476925287j)
(-0.05624371649767405067259453 - 9.424777960769379715387 93j)
```

Some special values:

```
>>> loggamma(1); loggamma(2)
0.0
0.0
>>> loggamma(3); +ln2
0.6931471805599453094172321
0.6931471805599453094172321
>>> loggamma(3.5); log(15*sqrt(pi)/8)
1.200973602347074224816022
1.200973602347074224816022
>>> loggamma(inf)
+inf
```

Huge arguments are permitted:

```
>>> loggamma('1e30')
6.807755278982137052053974e+31
>>> loggamma('1e300')
6.897755278982137052053974e+302
>>> loggamma('1e3000')
```

```
6.906755278982137052053974e+3003
>>> loggamma('1e10000000000000000000')
2.302585092994045684007991e+10000000000000000020
>>> loggamma('1e30j')
(-1.570796326794896619231322e+30 + 6.807755278982137052053974e+31j)
>>> loggamma('1e300j')
(-1.570796326794896619231322e+300 + 6.897755278982137052053974e+302j)
>>> loggamma('1e3000j')
(-1.570796326794896619231322e+3000 + 6.906755278982137052053974e+3003j)
```

The log-gamma function can be integrated analytically on any interval of unit length:

```
>>> z = 0
>>> quad(loggamma, [z,z+1]); log(2*pi)/2
0.918938533204672741780329 7
0.918938533204672741780329 7
>>> z = 3+4j
>>> quad(loggamma, [z,z+1]); (log(z)-1)*z + log(2*pi)/2
(-0.961928601499475064131442 1 + 5.219637303741238195688575j)
(-0.961928601499475064131442 1 + 5.219637303741238195688575j)
```

The derivatives of the log-gamma function are given by the polygamma function (`psi()` (page 698)):

```
>>> diff(loggamma, -4+3j); psi(0, -4+3j)
(1.688493531222971393607153 + 2.554898911356806978892748j)
(1.688493531222971393607153 + 2.554898911356806978892748j)
>>> diff(loggamma, -4+3j, 2); psi(1, -4+3j)
(-0.153941482921988237156103 8 - 0.102048519743026771974647 9j)
(-0.153941482921988237156103 8 - 0.102048519743026771974647 9j)
```

The log-gamma function satisfies an additive form of the recurrence relation for the ordinary gamma function:

```
>>> z = 2+3j
>>> loggamma(z); loggamma(z+1) - log(z)
(-2.092851753092733349564189 + 2.302396543466867626153708j)
(-2.092851753092733349564189 + 2.302396543466867626153708j)
```

Rising and falling factorials

rf()

mpmath.**rf**(*x*, *n*)

Computes the rising factorial or Pochhammer symbol,

$$x^{(n)} = x(x+1)\cdots(x+n-1) = \frac{\Gamma(x+n)}{\Gamma(x)}$$

where the rightmost expression is valid for nonintegral n.

Examples

For integral n, the rising factorial is a polynomial:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
```

```
...     nprint(taylor(lambda x: rf(x,n), 0, n))
...
[1.0]
[0.0, 1.0]
[0.0, 1.0, 1.0]
[0.0, 2.0, 3.0, 1.0]
[0.0, 6.0, 11.0, 6.0, 1.0]
```

Evaluation is supported for arbitrary arguments:

```
>>> rf(2+3j, 5.5)
(-7202.03920483347 - 3777.58810701527j)
```

ff()

mpmath.**ff**(*x*, *n*)

Computes the falling factorial,

$$(x)_n = x(x-1)\cdots(x-n+1) = \frac{\Gamma(x+1)}{\Gamma(x-n+1)}$$

where the rightmost expression is valid for nonintegral n.

Examples

For integral n, the falling factorial is a polynomial:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(taylor(lambda x: ff(x,n), 0, n))
...
[1.0]
[0.0, 1.0]
[0.0, -1.0, 1.0]
[0.0, 2.0, -3.0, 1.0]
[0.0, -6.0, 11.0, -6.0, 1.0]
```

Evaluation is supported for arbitrary arguments:

```
>>> ff(2+3j, 5.5)
(-720.41085888203 + 316.101124983878j)
```

Beta function

beta()

mpmath.**beta**(*x*, *y*)

Computes the beta function, $B(x,y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$. The beta function is also commonly defined by the integral representation

$$B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$$

Examples

For integer and half-integer arguments where all three gamma functions are finite, the beta function becomes either rational number or a rational multiple of π:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> beta(5, 2)
0.0333333333333333
>>> beta(1.5, 2)
0.266666666666667
>>> 16*beta(2.5, 1.5)
3.14159265358979
```

Where appropriate, `beta()` (page 691) evaluates limits. A pole of the beta function is taken to result in `+inf`:

```
>>> beta(-0.5, 0.5)
0.0
>>> beta(-3, 3)
-0.333333333333333
>>> beta(-2, 3)
+inf
>>> beta(inf, 1)
0.0
>>> beta(inf, 0)
nan
```

`beta()` (page 691) supports complex numbers and arbitrary precision evaluation:

```
>>> beta(1, 2+j)
(0.4 - 0.2j)
>>> mp.dps = 25
>>> beta(j,0.5)
(1.079424249270925780135675 - 1.410032405664160838288752j)
>>> mp.dps = 50
>>> beta(pi, e)
0.037890298781212201348153837138927165984170287886464
```

Various integrals can be computed by means of the beta function:

```
>>> mp.dps = 15
>>> quad(lambda t: t**2.5*(1-t)**2, [0, 1])
0.0230880230880231
>>> beta(3.5, 3)
0.0230880230880231
>>> quad(lambda t: sin(t)**4 * sqrt(cos(t)), [0, pi/2])
0.319504062596158
>>> beta(2.5, 0.75)/2
0.319504062596158
```

betainc()

mpmath.**betainc**(*a, b, x1=0, x2=1, regularized=False*)

`betainc(a, b, x1=0, x2=1, regularized=False)` gives the generalized incomplete beta function,

$$I_{x_1}^{x_2}(a,b) = \int_{x_1}^{x_2} t^{a-1}(1-t)^{b-1}dt.$$

When $x_1 = 0, x_2 = 1$, this reduces to the ordinary (complete) beta function $B(a,b)$; see `beta()` (page 691).

With the keyword argument `regularized=True`, `betainc()` (page 692) computes the regularized incomplete beta function $I_{x_1}^{x_2}(a,b)/B(a,b)$. This is the cumulative distribution of the beta distribution with parameters a, b.

Examples

Verifying that `betainc()` (page 692) computes the integral in the definition:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> x,y,a,b = 3, 4, 0, 6
>>> betainc(x, y, a, b)
-4010.4
>>> quad(lambda t: t**(x-1) * (1-t)**(y-1), [a, b])
-4010.4
```

The arguments may be arbitrary complex numbers:

```
>>> betainc(0.75, 1-4j, 0, 2+3j)
(0.2241657956955709603655887 + 0.3619619242700451992411724j)
```

With regularization:

```
>>> betainc(1, 2, 0, 0.25, regularized=True)
0.4375
>>> betainc(pi, e, 0, 1, regularized=True)   # Complete
1.0
```

The beta integral satisfies some simple argument transformation symmetries:

```
>>> mp.dps = 15
>>> betainc(2,3,4,5), -betainc(2,3,5,4), betainc(3,2,1-5,1-4)
(56.0833333333333, 56.0833333333333, 56.0833333333333)
```

The beta integral can often be evaluated analytically. For integer and rational arguments, the incomplete beta function typically reduces to a simple algebraic-logarithmic expression:

```
>>> mp.dps = 25
>>> identify(chop(betainc(0, 0, 3, 4)))
'-(log((9/8)))'
>>> identify(betainc(2, 3, 4, 5))
'(673/12)'
>>> identify(betainc(1.5, 1, 1, 2))
'((-12+sqrt(1152))/18)'
```

Super- and hyperfactorials

superfac()

mpmath.**superfac**(z)

Computes the superfactorial, defined as the product of consecutive factorials

$$\mathrm{sf}(n) = \prod_{k=1}^{n} k!$$

For general complex z, $\mathrm{sf}(z)$ is defined in terms of the Barnes G-function (see `barnesg()` (page 696)).

Examples

The first few superfactorials are (OEIS A000178):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(10):
...     print("%s %s" % (n, superfac(n)))
...
0 1.0
1 1.0
2 2.0
3 12.0
4 288.0
5 34560.0
6 24883200.0
7 125411328000.0
8 5.05658474496e+15
9 1.83493347225108e+21
```

Superfactorials grow very rapidly:

```
>>> superfac(1000)
3.24570818422368e+1177245
>>> superfac(10**10)
2.61398543581249e+467427913956904067453
```

Evaluation is supported for arbitrary arguments:

```
>>> mp.dps = 25
>>> superfac(pi)
17.20051550121297985285333
>>> superfac(2+3j)
(-0.00591548563319978962746646 8 + 0.00815644946460404494873826 3j)
>>> diff(superfac, 1)
0.2645072034016070205673056
```

References

1. http://www.research.att.com/~njas/sequences/A000178

hyperfac()

mpmath.**hyperfac**(*z*)

Computes the hyperfactorial, defined for integers as the product

$$H(n) = \prod_{k=1}^{n} k^k.$$

The hyperfactorial satisfies the recurrence formula $H(z) = z^z H(z-1)$. It can be defined more generally in terms of the Barnes G-function (see `barnesg()` (page 696)) and the gamma function by the formula

$$H(z) = \frac{\Gamma(z+1)^z}{G(z)}.$$

The extension to complex numbers can also be done via the integral representation

$$H(z) = (2\pi)^{-z/2} \exp\left[\binom{z+1}{2} + \int_0^z \log(t!)\,dt\right].$$

Examples

The rapidly-growing sequence of hyperfactorials begins (OEIS A002109):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(10):
...     print("%s %s" % (n, hyperfac(n)))
...
0 1.0
1 1.0
2 4.0
3 108.0
4 27648.0
5 86400000.0
6 4031078400000.0
7 3.3197663987712e+18
8 5.56964379417266e+25
9 2.15779412229419e+34
```

Some even larger hyperfactorials are:

```
>>> hyperfac(1000)
5.46458120882585e+1392926
>>> hyperfac(10**10)
4.60408207642219e+489142638002418704309
```

The hyperfactorial can be evaluated for arbitrary arguments:

```
>>> hyperfac(0.5)
0.880449235173423
>>> diff(hyperfac, 1)
0.581061466795327
>>> hyperfac(pi)
205.211134637462
>>> hyperfac(-10+1j)
(3.01144471378225e+46 - 2.45285242480185e+46j)
```

The recurrence property of the hyperfactorial holds generally:

```
>>> z = 3-4*j
>>> hyperfac(z)
(-4.49795891462086e-7 - 6.33262283196162e-7j)
>>> z**z * hyperfac(z-1)
(-4.49795891462086e-7 - 6.33262283196162e-7j)
>>> z = mpf(-0.6)
>>> chop(z**z * hyperfac(z-1))
1.28170142849352
>>> hyperfac(z)
1.28170142849352
```

The hyperfactorial may also be computed using the integral definition:

```
>>> z = 2.5
>>> hyperfac(z)
15.9842119922237
>>> (2*pi)**(-z/2)*exp(binomial(z+1,2) +
...     quad(lambda t: loggamma(t+1), [0, z]))
15.9842119922237
```

`hyperfac()` (page 694) supports arbitrary-precision evaluation:

```
>>> mp.dps = 50
>>> hyperfac(10)
215779412229418562091680268288000000000000000.0
>>> hyperfac(1/sqrt(2))
0.89404818005227001975423476035729076375705084390942
```

References

1. http://www.research.att.com/~njas/sequences/A002109
2. http://mathworld.wolfram.com/Hyperfactorial.html

barnesg()

mpmath.**barnesg**(z)

Evaluates the Barnes G-function, which generalizes the superfactorial (superfac() (page 693)) and by extension also the hyperfactorial (hyperfac() (page 694)) to the complex numbers in an analogous way to how the gamma function generalizes the ordinary factorial.

The Barnes G-function may be defined in terms of a Weierstrass product:

$$G(z+1) = (2\pi)^{z/2} e^{-[z(z+1)+\gamma z^2]/2} \prod_{n=1}^{\infty} \left[\left(1+\frac{z}{n}\right)^n e^{-z+z^2/(2n)}\right]$$

For positive integers n, we have have relation to superfactorials $G(n) = \mathrm{sf}(n-2) = 0! \cdot 1! \cdots (n-2)!$.

Examples

Some elementary values and limits of the Barnes G-function:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> barnesg(1), barnesg(2), barnesg(3)
(1.0, 1.0, 1.0)
>>> barnesg(4)
2.0
>>> barnesg(5)
12.0
>>> barnesg(6)
288.0
>>> barnesg(7)
34560.0
>>> barnesg(8)
24883200.0
>>> barnesg(inf)
+inf
>>> barnesg(0), barnesg(-1), barnesg(-2)
(0.0, 0.0, 0.0)
```

Closed-form values are known for some rational arguments:

```
>>> barnesg('1/2')
0.603244281209446
>>> sqrt(exp(0.25+log(2)/12)/sqrt(pi)/glaisher**3)
0.603244281209446
>>> barnesg('1/4')
0.29375596533861
>>> nthroot(exp('3/8')/exp(catalan/pi)/
```

```
...      gamma(0.25)**3/sqrt(glaisher)**9, 4)
0.29375596533861
```

The Barnes G-function satisfies the functional equation $G(z+1) = \Gamma(z)G(z)$:

```
>>> z = pi
>>> barnesg(z+1)
2.39292119327948
>>> gamma(z)*barnesg(z)
2.39292119327948
```

The asymptotic growth rate of the Barnes G-function is related to the Glaisher-Kinkelin constant:

```
>>> limit(lambda n: barnesg(n+1)/(n**(n**2/2-mpf(1)/12)*
...     (2*pi)**(n/2)*exp(-3*n**2/4)), inf)
0.847536694177301
>>> exp('1/12')/glaisher
0.847536694177301
```

The Barnes G-function can be differentiated in closed form:

```
>>> z = 3
>>> diff(barnesg, z)
0.264507203401607
>>> barnesg(z)*((z-1)*psi(0,z)-z+(log(2*pi)+1)/2)
0.264507203401607
```

Evaluation is supported for arbitrary arguments and at arbitrary precision:

```
>>> barnesg(6.5)
2548.7457695685
>>> barnesg(-pi)
0.00535976768353037
>>> barnesg(3+4j)
(-0.000676375932234244 - 4.42236140124728e-5j)
>>> mp.dps = 50
>>> barnesg(1/sqrt(2))
0.81305501090451340843586085064413533788206204124732
>>> q = barnesg(10j)
>>> q.real
0.000000000021852360840356557241543036724799812371995850552234
>>> q.imag
-0.000000000000700353353200623048490206542155458390532100414575 88
>>> mp.dps = 15
>>> barnesg(100)
3.10361006263698e+6626
>>> barnesg(-101)
0.0
>>> barnesg(-10.5)
5.94463017605008e+25
>>> barnesg(-10000.5)
-6.14322868174828e+167480422
>>> barnesg(1000j)
(5.21133054865546e-1173597 + 4.27461836811016e-1173597j)
>>> barnesg(-1000+1000j)
(2.43114569750291e+1026623 + 2.24851410674842e+1026623j)
```

References

1. Whittaker & Watson, *A Course of Modern Analysis*, Cambridge University Press, 4th edition (1927), p.264
2. http://en.wikipedia.org/wiki/Barnes_G-function
3. http://mathworld.wolfram.com/BarnesG-Function.html

Polygamma functions and harmonic numbers

psi()/digamma()

mpmath.**psi**(*m*, *z*)

Gives the polygamma function of order m of z, $\psi^{(m)}(z)$. Special cases are known as the *digamma function* ($\psi^{(0)}(z)$), the *trigamma function* ($\psi^{(1)}(z)$), etc. The polygamma functions are defined as the logarithmic derivatives of the gamma function:

$$\psi^{(m)}(z) = \left(\frac{d}{dz}\right)^{m+1} \log \Gamma(z)$$

In particular, $\psi^{(0)}(z) = \Gamma'(z)/\Gamma(z)$. In the present implementation of psi() (page 698), the order m must be a nonnegative integer, while the argument z may be an arbitrary complex number (with exception for the polygamma function's poles at $z = 0, -1, -2, \ldots$).

Examples

For various rational arguments, the polygamma function reduces to a combination of standard mathematical constants:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> psi(0, 1), -euler
(-0.5772156649015328606065121, -0.5772156649015328606065121)
>>> psi(1, '1/4'), pi**2+8*catalan
(17.19732915450711073927132, 17.19732915450711073927132)
>>> psi(2, '1/2'), -14*apery
(-16.828796644234319995596 33, -16.82879664423431999559633)
```

The polygamma functions are derivatives of each other:

```
>>> diff(lambda x: psi(3, x), pi), psi(4, pi)
(-0.11057493125788627345269 52, -0.11057493125788627345269 52)
>>> quad(lambda x: psi(4, x), [2, 3]), psi(3,3)-psi(3,2)
(-0.375, -0.375)
```

The digamma function diverges logarithmically as $z \to \infty$, while higher orders tend to zero:

```
>>> psi(0,inf), psi(1,inf), psi(2,inf)
(+inf, 0.0, 0.0)
```

Evaluation for a complex argument:

```
>>> psi(2, -1-2j)
(0.03902435405364952654838445 + 0.1574325240413029954685366j)
```

Evaluation is supported for large orders m and/or large arguments z:

```
>>> psi(3, 10**100)
2.0e-300
>>> psi(250, 10**30+10**20*j)
(-1.29314250436364268720486 5e-7010 + 3.23285626090910739151310 8e-7018j)
```

Application to infinite series

Any infinite series where the summand is a rational function of the index k can be evaluated in closed form in terms of polygamma functions of the roots and poles of the summand:

```
>>> a = sqrt(2)
>>> b = sqrt(3)
>>> nsum(lambda k: 1/((k+a)**2*(k+b)), [0, inf])
0.404966892751785706191753 1
>>> (psi(0,a)-psi(0,b)-a*psi(1,a)+b*psi(1,a))/(a-b)**2
0.404966892751785706191753 1
```

This follows from the series representation ($m > 0$)

$$\psi^{(m)}(z) = (-1)^{m+1} m! \sum_{k=0}^{\infty} \frac{1}{(z+k)^{m+1}}.$$

Since the roots of a polynomial may be complex, it is sometimes necessary to use the complex polygamma function to evaluate an entirely real-valued sum:

```
>>> nsum(lambda k: 1/(k**2-2*k+3), [0, inf])
1.69436143390706125615466 5
>>> nprint(polyroots([1,-2,3]))
[(1.0 - 1.41421j), (1.0 + 1.41421j)]
>>> r1 = 1-sqrt(2)*j
>>> r2 = r1.conjugate()
>>> (psi(0,-r2)-psi(0,-r1))/(r1-r2)
(1.69436143390706125615466 5 + 0.0j)
```

mpmath.**digamma**(z)

Shortcut for `psi(0,z)`.

harmonic()

mpmath.**harmonic**(z)

If n is an integer, `harmonic(n)` gives a floating-point approximation of the n-th harmonic number $H(n)$, defined as

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

The first few harmonic numbers are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(8):
...     print("%s %s" % (n, harmonic(n)))
...
0 0.0
1 1.0
2 1.5
3 1.83333333333333
4 2.08333333333333
```

```
5 2.28333333333333
6 2.45
7 2.59285714285714
```

The infinite harmonic series $1 + 1/2 + 1/3 + \ldots$ diverges:

```
>>> harmonic(inf)
+inf
```

harmonic() (page 699) is evaluated using the digamma function rather than by summing the harmonic series term by term. It can therefore be computed quickly for arbitrarily large n, and even for nonintegral arguments:

```
>>> harmonic(10**100)
230.835724964306
>>> harmonic(0.5)
0.613705638880109
>>> harmonic(3+4j)
(2.24757548223494 + 0.850502209186044j)
```

harmonic() (page 699) supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> harmonic(11)
3.0198773448773448773448773448773448773448773448773
>>> harmonic(pi)
1.8727388590273302654363491032336134987519132374152
```

The harmonic series diverges, but at a glacial pace. It is possible to calculate the exact number of terms required before the sum exceeds a given amount, say 100:

```
>>> mp.dps = 50
>>> v = 10**findroot(lambda x: harmonic(10**x) - 100, 10)
>>> v
15092688622113788323693563264538101449859496.864101
>>> v = int(ceil(v))
>>> print(v)
15092688622113788323693563264538101449859497
>>> harmonic(v-1)
99.999999999999999999999999999999999999999942747
>>> harmonic(v)
100.000000000000000000000000000000000000000009
```

Exponential integrals and error functions

Exponential integrals give closed-form solutions to a large class of commonly occurring transcendental integrals that cannot be evaluated using elementary functions. Integrals of this type include those with an integrand of the form $t^a e^t$ or e^{-x^2}, the latter giving rise to the Gaussian (or normal) probability distribution.

The most general function in this section is the incomplete gamma function, to which all others can be reduced. The incomplete gamma function, in turn, can be expressed using hypergeometric functions (see *Hypergeometric functions* (page 793)).

Incomplete gamma functions

gammainc()

mpmath.**gammainc**(*z, a=0, b=inf, regularized=False*)

`gammainc(z, a=0, b=inf)` computes the (generalized) incomplete gamma function with integration limits $[a, b]$:

$$\Gamma(z, a, b) = \int_a^b t^{z-1} e^{-t}\, dt$$

The generalized incomplete gamma function reduces to the following special cases when one or both endpoints are fixed:

- $\Gamma(z, 0, \infty)$ is the standard ("complete") gamma function, $\Gamma(z)$ (available directly as the mpmath function `gamma()` (page 686))
- $\Gamma(z, a, \infty)$ is the "upper" incomplete gamma function, $\Gamma(z, a)$
- $\Gamma(z, 0, b)$ is the "lower" incomplete gamma function, $\gamma(z, b)$.

Of course, we have $\Gamma(z, 0, x) + \Gamma(z, x, \infty) = \Gamma(z)$ for all z and x.

Note however that some authors reverse the order of the arguments when defining the lower and upper incomplete gamma function, so one should be careful to get the correct definition.

If also given the keyword argument `regularized=True`, `gammainc()` (page 701) computes the "regularized" incomplete gamma function

$$P(z, a, b) = \frac{\Gamma(z, a, b)}{\Gamma(z)}.$$

Examples

We can compare with numerical quadrature to verify that `gammainc()` (page 701) computes the integral in the definition:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> gammainc(2+3j, 4, 10)
(0.009772126686277051606023 12 - 0.07706373063129898924519 77j)
>>> quad(lambda t: t**(2+3j-1) * exp(-t), [4, 10])
(0.009772126686277051606023 12 - 0.07706373063129898924519 77j)
```

Argument symmetries follow directly from the integral definition:

```
>>> gammainc(3, 4, 5) + gammainc(3, 5, 4)
0.0
>>> gammainc(3,0,2) + gammainc(3,2,4); gammainc(3,0,4)
1.523793388892911312363331
1.523793388892911312363331
>>> findroot(lambda z: gammainc(2,z,3), 1)
3.0
```

Evaluation for arbitrarily large arguments:

```
>>> gammainc(10, 100)
4.083660630910611272288592e-26
>>> gammainc(10, 10000000000000000)
5.290402449901174752972486e-4342944819032375
>>> gammainc(3+4j, 1000000+1000000j)
(-1.257913707524362408877881e-434284 + 2.556691003883483531962095e-434284j)
```

Evaluation of a generalized incomplete gamma function automatically chooses the representation that gives a more accurate result, depending on which parameter is larger:

```
>>> gammainc(10000000, 3) - gammainc(10000000, 2)   # Bad
0.0
>>> gammainc(10000000, 2, 3)   # Good
1.755146243738946045873491e+4771204
>>> gammainc(2, 0, 100000001) - gammainc(2, 0, 100000000)   # Bad
0.0
>>> gammainc(2, 100000000, 100000001)   # Good
4.078258353474186729184421e-43429441
```

The incomplete gamma functions satisfy simple recurrence relations:

```
>>> mp.dps = 25
>>> z, a = mpf(3.5), mpf(2)
>>> gammainc(z+1, a); z*gammainc(z,a) + a**z*exp(-a)
10.60130296933533459267329
10.60130296933533459267329
>>> gammainc(z+1,0,a); z*gammainc(z,0,a) - a**z*exp(-a)
1.030425427232114336470932
1.030425427232114336470932
```

Evaluation at integers and poles:

```
>>> gammainc(-3, -4, -5)
(-0.221457704896779856623419 2 + 0.0j)
>>> gammainc(-3, 0, 5)
+inf
```

If z is an integer, the recurrence reduces the incomplete gamma function to $P(a)\exp(-a)+Q(b)\exp(-b)$ where P and Q are polynomials:

```
>>> gammainc(1, 2); exp(-2)
0.1353352832366126918939995
0.1353352832366126918939995
>>> mp.dps = 50
>>> identify(gammainc(6, 1, 2), ['exp(-1)', 'exp(-2)'])
'(326*exp(-1) + (-872)*exp(-2))'
```

The incomplete gamma functions reduce to functions such as the exponential integral Ei and the error function for special arguments:

```
>>> mp.dps = 25
>>> gammainc(0, 4); -ei(-4)
0.0037793524098489064788748 6
0.0037793524098489064788748 6
>>> gammainc(0.5, 0, 2); sqrt(pi)*erf(sqrt(2))
1.691806732945198336509541
1.691806732945198336509541
```

Exponential integrals

ei()

mpmath.**ei**(*x*, ***kwargs*)

Computes the exponential integral or Ei-function, $\mathrm{Ei}(x)$. The exponential integral is de-

fined as

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t}\, dt.$$

When the integration range includes $t = 0$, the exponential integral is interpreted as providing the Cauchy principal value.

For real x, the Ei-function behaves roughly like $\mathrm{Ei}(x) \approx \exp(x) + \log(|x|)$.

The Ei-function is related to the more general family of exponential integral functions denoted by E_n, which are available as `expint()` (page 704).

Basic examples

Some basic values and limits are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> ei(0)
-inf
>>> ei(1)
1.89511781635594
>>> ei(inf)
+inf
>>> ei(-inf)
0.0
```

For $x < 0$, the defining integral can be evaluated numerically as a reference:

```
>>> ei(-4)
-0.00377935240984891
>>> quad(lambda t: exp(t)/t, [-inf, -4])
-0.00377935240984891
```

`ei()` (page 702) supports complex arguments and arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> ei(pi)
10.928374389331410348638445906907535171566338835056
>>> mp.dps = 25
>>> ei(3+4j)
(-4.154091651642689822535359 + 4.294418620024357476985535j)
```

Related functions

The exponential integral is closely related to the logarithmic integral. See `li()` (page 705) for additional information.

The exponential integral is related to the hyperbolic and trigonometric integrals (see `chi()` (page 709), `shi()` (page 709), `ci()` (page 707), `si()` (page 708)) similarly to how the ordinary exponential function is related to the hyperbolic and trigonometric functions:

```
>>> mp.dps = 15
>>> ei(3)
9.93383257062542
>>> chi(3) + shi(3)
9.93383257062542
>>> chop(ci(3j) - j*si(3j) - pi*j/2)
9.93383257062542
```

Beware that logarithmic corrections, as in the last example above, are required to obtain the correct branch in general. For details, see [1].

The exponential integral is also a special case of the hypergeometric function ${}_2F_2$:

```
>>> z = 0.6
>>> z*hyper([1,1],[2,2],z) + (ln(z)-ln(1/z))/2 + euler
0.769881289937359
>>> ei(z)
0.769881289937359
```

References

1. Relations between Ei and other functions: http://functions.wolfram.com/GammaBetaErf/ExpIntegr
2. Abramowitz & Stegun, section 5: http://www.math.sfu.ca/~cbm/aands/page_228.htm
3. Asymptotic expansion for Ei: http://mathworld.wolfram.com/En-Function.html

e1()

mpmath.**e1**(*x*, ***kwargs*)

Computes the exponential integral $\mathrm{E}_1(z)$, given by

$$\mathrm{E}_1(z) = \int_z^\infty \frac{e^{-t}}{t} dt.$$

This is equivalent to `expint()` (page 704) with $n = 1$.

Examples

Two ways to evaluate this function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> e1(6.25)
0.0002704758872637179088496194
>>> expint(1,6.25)
0.0002704758872637179088496194
```

The E1-function is essentially the same as the Ei-function (`ei()` (page 702)) with negated argument, except for an imaginary branch cut term:

```
>>> e1(2.5)
0.02491491787026973549562801
>>> -ei(-2.5)
0.02491491787026973549562801
>>> e1(-2.5)
(-7.07376589457860071192355 2 - 3.14159265358979323846264 3j)
>>> -ei(2.5)
-7.07376589457860071192355 2
```

expint()

mpmath.**expint**(**args*)

`expint(n,z)()` gives the generalized exponential integral or En-function,

$$\mathrm{E}_n(z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt,$$

where n and z may both be complex numbers. The case with $n = 1$ is also given by `e1()` (page 704).

Examples

Evaluation at real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> expint(1, 6.25)
0.0002704758872637179088496194
>>> expint(-3, 2+3j)
(0.00299658467335472929656159 + 0.0610081620212588545031963 2j)
>>> expint(2+3j, 4-5j)
(0.001803529474663565056945248 - 0.002235061547756185403349091j)
```

At negative integer values of n, $E_n(z)$ reduces to a rational-exponential function:

```
>>> f = lambda n, z: fac(n)*sum(z**k/fac(k-1) for k in range(1,n+2))/\
...     exp(z)/z**(n+2)
>>> n = 3
>>> z = 1/pi
>>> expint(-n,z)
584.2604820613019908668219
>>> f(n,z)
584.2604820613019908668219
>>> n = 5
>>> expint(-n,z)
115366.5762594725451811138
>>> f(n,z)
115366.5762594725451811138
```

Logarithmic integral

li()

mpmath.**li**(*x*, ***kwargs*)

Computes the logarithmic integral or li-function $\mathrm{li}(x)$, defined by

$$\mathrm{li}(x) = \int_0^x \frac{1}{\log t}\,dt$$

The logarithmic integral has a singularity at $x = 1$.

Alternatively, `li(x, offset=True)` computes the offset logarithmic integral (used in number theory)

$$\mathrm{Li}(x) = \int_2^x \frac{1}{\log t}\,dt.$$

These two functions are related via the simple identity $\mathrm{Li}(x) = \mathrm{li}(x) - \mathrm{li}(2)$.

The logarithmic integral should also not be confused with the polylogarithm (also denoted by Li), which is implemented as `polylog()` (page 849).

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> li(0)
0.0
```

```
>>> li(1)
-inf
>>> li(1)
-inf
>>> li(2)
1.0451637801174927848445888919
>>> findroot(li, 2)
1.4513692348833810502839684858 9
>>> li(inf)
+inf
>>> li(2, offset=True)
0.0
>>> li(1, offset=True)
-inf
>>> li(0, offset=True)
-1.0451637801174927848445888919
>>> li(10, offset=True)
5.1204357246698051526783928634 7
```

The logarithmic integral can be evaluated for arbitrary complex arguments:

```
>>> mp.dps = 20
>>> li(3+4j)
(3.1343755504645775265 + 2.6769247817778742392j)
```

The logarithmic integral is related to the exponential integral:

```
>>> ei(log(3))
2.1635885946671919729
>>> li(3)
2.1635885946671919729
```

The logarithmic integral grows like $O(x/\log(x))$:

```
>>> mp.dps = 15
>>> x = 10**100
>>> x/log(x)
4.34294481903252e+97
>>> li(x)
4.3619719871407e+97
```

The prime number theorem states that the number of primes less than x is asymptotic to $\mathrm{Li}(x)$ (equivalently $\mathrm{li}(x)$). For example, it is known that there are exactly 1,925,320,391,606,803,968,923 prime numbers less than 10^{23} [1]. The logarithmic integral provides a very accurate estimate:

```
>>> li(10**23, offset=True)
1.92532039161405e+21
```

A definite integral is:

```
>>> quad(li, [0, 1])
-0.693147180559945
>>> -ln(2)
-0.693147180559945
```

References

1. http://mathworld.wolfram.com/PrimeCountingFunction.html
2. http://mathworld.wolfram.com/LogarithmicIntegral.html

Trigonometric integrals

ci()
mpmath.**ci**(*x*, ***kwargs*)

Computes the cosine integral,

$$\mathrm{Ci}(x) = -\int_x^\infty \frac{\cos t}{t}\,dt = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t}\,dt$$

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ci(0)
-inf
>>> ci(1)
0.3374039229009681346626462
>>> ci(pi)
0.07366791204642548599010096
>>> ci(inf)
0.0
>>> ci(-inf)
(0.0 + 3.141592653589793238462643j)
>>> ci(2+3j)
(1.408292501520849518759125 - 2.983617742029605093121118j)
```

The cosine integral behaves roughly like the sinc function (see `sinc()` (page 679)) for large real x:

```
>>> ci(10**10)
-4.875060251748226537857298e-11
>>> sinc(10**10)
-4.875060250875106915277943e-11
>>> chop(limit(ci, inf))
0.0
```

It has infinitely many roots on the positive real axis:

```
>>> findroot(ci, 1)
0.6165054856207162337971104
>>> findroot(ci, 2)
3.384180422551186426397851
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> ci(10**6*(1+j))
(4.449410587611035724984376e+434287 + 9.757448742900135264170 59e+434287j)
```

We can evaluate the defining integral as a reference:

```
>>> mp.dps = 15
>>> -quadosc(lambda t: cos(t)/t, [5, inf], omega=1)
-0.190029749656644
>>> ci(5)
-0.190029749656644
```

Some infinite series can be evaluated using the cosine integral:

```
>>> nsum(lambda k: (-1)**k/(fac(2*k)*(2*k)), [1,inf])
-0.239811742000565
>>> ci(1) - euler
-0.239811742000565
```

si()

mpmath.**si**(*x*, ***kwargs*)

Computes the sine integral,

$$\mathrm{Si}(x) = \int_0^x \frac{\sin t}{t}\, dt.$$

The sine integral is thus the antiderivative of the sinc function (see `sinc()` (page 679)).

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> si(0)
0.0
>>> si(1)
0.9460830703671830149413533
>>> si(-1)
-0.9460830703671830149413533
>>> si(pi)
1.851937051982466170361053
>>> si(inf)
1.570796326794896619231322
>>> si(-inf)
-1.570796326794896619231322
>>> si(2+3j)
(4.547513889562289219853204 + 1.399196580646054789459839j)
```

The sine integral approaches $\pi/2$ for large real x:

```
>>> si(10**10)
1.570796326707584656968511
>>> pi/2
1.570796326794896619231322
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> si(10**6*(1+j))
(-9.757448742900135264170 59e+434287 + 4.449410587611035724984376e+434287j)
```

We can evaluate the defining integral as a reference:

```
>>> mp.dps = 15
>>> quad(sinc, [0, 5])
1.54993124494467
>>> si(5)
1.54993124494467
```

Some infinite series can be evaluated using the sine integral:

```
>>> nsum(lambda k: (-1)**k/(fac(2*k+1)*(2*k+1)), [0,inf])
0.946083070367183
```

```
>>> si(1)
0.946083070367183
```

Hyperbolic integrals

chi()

mpmath.**chi**(*x*, ***kwargs*)

Computes the hyperbolic cosine integral, defined in analogy with the cosine integral (see ci() (page 707)) as

$$\operatorname{Chi}(x) = -\int_x^\infty \frac{\cosh t}{t}\,dt = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t}\,dt$$

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> chi(0)
-inf
>>> chi(1)
0.8378669409802082408946786
>>> chi(inf)
+inf
>>> findroot(chi, 0.5)
0.5238225713898644064509583
>>> chi(2+3j)
(-0.1683628683277204662429321 + 2.625115880451325002151688j)
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> chi(10**6*(1+j))
(4.449410587611035724984376e+434287 - 9.757448742900135264170 59e+434287j)
```

shi()

mpmath.**shi**(*x*, ***kwargs*)

Computes the hyperbolic sine integral, defined in analogy with the sine integral (see si() (page 708)) as

$$\operatorname{Shi}(x) = \int_0^x \frac{\sinh t}{t}\,dt.$$

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> shi(0)
0.0
>>> shi(1)
1.057250875375728514571842
>>> shi(-1)
-1.057250875375728514571842
>>> shi(inf)
+inf
>>> shi(2+3j)
(-0.193189076271919829167809 5 + 2.645432555362369624818525j)
```

Evaluation is supported for z anywhere in the complex plane:

```
>>> shi(10**6*(1+j))
(4.449410587611035724984376e+434287 - 9.757448742900135264170 59e+434287j)
```

Error functions

erf()

mpmath.**erf**(*x*, ***kwargs*)

Computes the error function, $\mathrm{erf}(x)$. The error function is the normalized antiderivative of the Gaussian function $\exp(-t^2)$. More precisely,

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\,dt$$

Basic examples

Simple values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> erf(0)
0.0
>>> erf(1)
0.842700792949715
>>> erf(-1)
-0.842700792949715
>>> erf(inf)
1.0
>>> erf(-inf)
-1.0
```

For large real x, $\mathrm{erf}(x)$ approaches 1 very rapidly:

```
>>> erf(3)
0.999977909503001
>>> erf(5)
0.999999999998463
```

The error function is an odd function:

```
>>> nprint(chop(taylor(erf, 0, 5)))
[0.0, 1.12838, 0.0, -0.376126, 0.0, 0.112838]
```

`erf()` (page 710) implements arbitrary-precision evaluation and supports complex numbers:

```
>>> mp.dps = 50
>>> erf(0.5)
0.52049987781304653768274665389196452873645157575796
>>> mp.dps = 25
>>> erf(1+j)
(1.316151281697947644880271 + 0.190453469237834686284108 9j)
```

Evaluation is supported for large arguments:

```
>>> mp.dps = 25
>>> erf('1e1000')
1.0
>>> erf('-1e1000')
-1.0
>>> erf('1e-1000')
1.128379167095512573896159e-1000
>>> erf('1e7j')
(0.0 + 8.593897639029319267398803e+43429448190317j)
>>> erf('1e7+1e7j')
(0.9999999858172446172631323 + 3.728805278735270407053139e-8j)
```

Related functions

See also `erfc()` (page 711), which is more accurate for large x, and `erfi()` (page 711) which gives the antiderivative of $\exp(t^2)$.

The Fresnel integrals `fresnels()` (page 714) and `fresnelc()` (page 715) are also related to the error function.

erfc()

mpmath.**erfc**(x, ***kwargs*)

Computes the complementary error function, $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$. This function avoids cancellation that occurs when naively computing the complementary error function as `1-erf(x)`:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> 1 - erf(10)
0.0
>>> erfc(10)
2.08848758376254e-45
```

`erfc()` (page 711) works accurately even for ludicrously large arguments:

```
>>> erfc(10**10)
4.3504398860243e-43429448190325182776
```

Complex arguments are supported:

```
>>> erfc(500+50j)
(1.19739830969552e-107492 + 1.46072418957528e-107491j)
```

erfi()

mpmath.**erfi**(x)

Computes the imaginary error function, $\operatorname{erfi}(x)$. The imaginary error function is defined in analogy with the error function, but with a positive sign in the integrand:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(t^2)\, dt$$

Whereas the error function rapidly converges to 1 as x grows, the imaginary error function rapidly diverges to infinity. The functions are related as $\operatorname{erfi}(x) = -i \operatorname{erf}(ix)$ for all complex numbers x.

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> erfi(0)
0.0
>>> erfi(1)
1.65042575879754
>>> erfi(-1)
-1.65042575879754
>>> erfi(inf)
+inf
>>> erfi(-inf)
-inf
```

Note the symmetry between erf and erfi:

```
>>> erfi(3j)
(0.0 + 0.999977909503001j)
>>> erf(3)
0.999977909503001
>>> erf(1+2j)
(-0.536643565778565 - 5.04914370344703j)
>>> erfi(2+1j)
(-5.04914370344703 - 0.536643565778565j)
```

Large arguments are supported:

```
>>> erfi(1000)
1.71130938718796e+434291
>>> erfi(10**10)
7.3167287567024e+43429448190325182754
>>> erfi(-10**10)
-7.3167287567024e+43429448190325182754
>>> erfi(1000-500j)
(2.49895233563961e+325717 + 2.6846779342253e+325717j)
>>> erfi(100000j)
(0.0 + 1.0j)
>>> erfi(-100000j)
(0.0 - 1.0j)
```

erfinv()

mpmath.**erfinv**(x)

Computes the inverse error function, satisfying

$$\operatorname{erf}(\operatorname{erfinv}(x)) = \operatorname{erfinv}(\operatorname{erf}(x)) = x.$$

This function is defined only for $-1 \le x \le 1$.

Examples

Special values include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> erfinv(0)
0.0
>>> erfinv(1)
+inf
>>> erfinv(-1)
-inf
```

The domain is limited to the standard interval:

```
>>> erfinv(2)
Traceback (most recent call last):
  ...
ValueError: erfinv(x) is defined only for -1 <= x <= 1
```

It is simple to check that erfinv() (page 712) computes inverse values of erf() (page 710) as promised:

```
>>> erf(erfinv(0.75))
0.75
>>> erf(erfinv(-0.995))
-0.995
```

erfinv() (page 712) supports arbitrary-precision evaluation:

```
>>> mp.dps = 50
>>> x = erf(2)
>>> x
0.99532226501895273416206925636725292861089179704006
>>> erfinv(x)
2.0
```

A definite integral involving the inverse error function:

```
>>> mp.dps = 15
>>> quad(erfinv, [0, 1])
0.564189583547756
>>> 1/sqrt(pi)
0.564189583547756
```

The inverse error function can be used to generate random numbers with a Gaussian distribution (although this is a relatively inefficient algorithm):

```
>>> nprint([erfinv(2*rand()-1) for n in range(6)])
[-0.586747, 1.10233, -0.376796, 0.926037, -0.708142, -0.732012]
```

The normal distribution

npdf()

mpmath.**npdf**(*x, mu=0, sigma=1*)

npdf(x, mu=0, sigma=1) evaluates the probability density function of a normal distribution with mean value μ and variance σ^2.

Elementary properties of the probability distribution can be verified using numerical integration:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> quad(npdf, [-inf, inf])
1.0
>>> quad(lambda x: npdf(x, 3), [3, inf])
0.5
>>> quad(lambda x: npdf(x, 3, 2), [3, inf])
0.5
```

See also ncdf() (page 714), which gives the cumulative distribution.

ncdf()

mpmath.**ncdf**(*x, mu=0, sigma=1*)

`ncdf(x, mu=0, sigma=1)` evaluates the cumulative distribution function of a normal distribution with mean value μ and variance σ^2.

See also `npdf()` (page 713), which gives the probability density.

Elementary properties include:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> ncdf(pi, mu=pi)
0.5
>>> ncdf(-inf)
0.0
>>> ncdf(+inf)
1.0
```

The cumulative distribution is the integral of the density function having identical mu and sigma:

```
>>> mp.dps = 15
>>> diff(ncdf, 2)
0.053990966513188
>>> npdf(2)
0.053990966513188
>>> diff(lambda x: ncdf(x, 1, 0.5), 0)
0.107981933026376
>>> npdf(0, 1, 0.5)
0.107981933026376
```

Fresnel integrals

fresnels()

mpmath.**fresnels**(*x*)

Computes the Fresnel sine integral

$$S(x) = \int_0^x \sin\left(\frac{\pi t^2}{2}\right) dt$$

Note that some sources define this function without the normalization factor $\pi/2$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> fresnels(0)
0.0
>>> fresnels(inf)
0.5
>>> fresnels(-inf)
-0.5
>>> fresnels(1)
0.4382591473903547660767567
>>> fresnels(1+2j)
(36.72546488399143842838788 + 15.58775110440458732748279j)
```

Comparing with the definition:

```
>>> fresnels(3)
0.49631299896737503609761 23
>>> quad(lambda t: sin(pi*t**2/2), [0,3])
0.49631299896737503609761 23
```

fresnelc()

mpmath.**fresnelc**(*x*)

Computes the Fresnel cosine integral

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right) dt$$

Note that some sources define this function without the normalization factor $\pi/2$.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> fresnelc(0)
0.0
>>> fresnelc(inf)
0.5
>>> fresnelc(-inf)
-0.5
>>> fresnelc(1)
0.7798934003768228294742064
>>> fresnelc(1+2j)
(16.08787137412548041729489 - 36.22568799288165021578758j)
```

Comparing with the definition:

```
>>> fresnelc(3)
0.60572078929768562955616 11
>>> quad(lambda t: cos(pi*t**2/2), [0,3])
0.60572078929768562955616 11
```

Bessel functions and related functions

The functions in this section arise as solutions to various differential equations in physics, typically describing wavelike oscillatory behavior or a combination of oscillation and exponential decay or growth. Mathematically, they are special cases of the confluent hypergeometric functions ${}_0F_1$, ${}_1F_1$ and ${}_1F_2$ (see *Hypergeometric functions* (page 793)).

Bessel functions

besselj()

mpmath.**besselj**(*n, x, derivative=0*)

`besselj(n, x, derivative=0)` gives the Bessel function of the first kind $J_n(x)$. Bessel functions of the first kind are defined as solutions of the differential equation

$$x^2 y'' + x y' + (x^2 - n^2) y = 0$$

which appears, among other things, when solving the radial part of Laplace's equation in cylindrical coordinates. This equation has two solutions for given n, where the J_n-function is the solution that is nonsingular at $x = 0$. For positive integer n, $J_n(x)$ behaves roughly like a sine (odd n) or cosine (even n) multiplied by a magnitude factor that decays slowly as $x \to \pm\infty$.

Generally, J_n is a special case of the hypergeometric function ${}_0F_1$:

$$J_n(x) = \frac{x^n}{2^n\Gamma(n+1)}\,{}_0F_1\left(n+1, -\frac{x^2}{4}\right)$$

With *derivative* = $m \neq 0$, the m-th derivative

$$\frac{d^m}{dx^m} J_n(x)$$

is computed.

Plots

```
# Bessel function J_n(x) on the real line for n=0,1,2,3
j0 = lambda x: besselj(0,x)
j1 = lambda x: besselj(1,x)
j2 = lambda x: besselj(2,x)
j3 = lambda x: besselj(3,x)
plot([j0,j1,j2,j3],[0,14])
```

```
# Bessel function J_n(z) in the complex plane
cplot(lambda z: besselj(1,z), [-8,8], [-8,8], points=50000)
```

Examples

Evaluation is supported for arbitrary arguments, and at arbitrary precision:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> besselj(2, 1000)
-0.024777229528606
>>> besselj(4, 0.75)
0.000801070086542314
>>> besselj(2, 1000j)
(-2.48071721019185e+432 + 6.41567059811949e-437j)
>>> mp.dps = 25
>>> besselj(0.75j, 3+4j)
(-2.778118364828153309919653 - 1.586360388901862158553 3j)
>>> mp.dps = 50
>>> besselj(1, pi)
0.28461534317975275734531059968613140570981118184947
```

Arguments may be large:

```
>>> mp.dps = 25
>>> besselj(0, 10000)
-0.007096160353388801477265164
>>> besselj(0, 10**10)
0.00000217559175024689172685905 5
>>> besselj(2, 10**100)
7.337048736538615712436929e-51
>>> besselj(2, 10**5*j)
```

```
(-3.54072541197094886017373 5e+43426 + 4.49498124096158031100 51e-43433j)
```

The Bessel functions of the first kind satisfy simple symmetries around $x = 0$:

```
>>> mp.dps = 15
>>> nprint([besselj(n,0) for n in range(5)])
[1.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint([besselj(n,pi) for n in range(5)])
[-0.304242, 0.284615, 0.485434, 0.333458, 0.151425]
>>> nprint([besselj(n,-pi) for n in range(5)])
[-0.304242, -0.284615, 0.485434, -0.333458, 0.151425]
```

Roots of Bessel functions are often used:

```
>>> nprint([findroot(j0, k) for k in [2, 5, 8, 11, 14]])
[2.40483, 5.52008, 8.65373, 11.7915, 14.9309]
>>> nprint([findroot(j1, k) for k in [3, 7, 10, 13, 16]])
[3.83171, 7.01559, 10.1735, 13.3237, 16.4706]
```

The roots are not periodic, but the distance between successive roots asymptotically approaches 2π. Bessel functions of the first kind have the following normalization:

```
>>> quadosc(j0, [0, inf], period=2*pi)
1.0
>>> quadosc(j1, [0, inf], period=2*pi)
1.0
```

For $n = 1/2$ or $n = -1/2$, the Bessel function reduces to a trigonometric function:

```
>>> x = 10
>>> besselj(0.5, x), sqrt(2/(pi*x))*sin(x)
(-0.13726373575505, -0.13726373575505)
>>> besselj(-0.5, x), sqrt(2/(pi*x))*cos(x)
(-0.211708866331398, -0.211708866331398)
```

Derivatives of any order can be computed (negative orders correspond to integration):

```
>>> mp.dps = 25
>>> besselj(0, 7.5, 1)
-0.1352484275797055051822405
>>> diff(lambda x: besselj(0,x), 7.5)
-0.1352484275797055051822405
>>> besselj(0, 7.5, 10)
-0.1377811164763244890135677
>>> diff(lambda x: besselj(0,x), 7.5, 10)
-0.1377811164763244890135677
>>> besselj(0,7.5,-1) - besselj(0,3.5,-1)
-0.1241343240399987693521378
>>> quad(j0, [3.5, 7.5])
-0.1241343240399987693521378
```

Differentiation with a noninteger order gives the fractional derivative in the sense of the Riemann-Liouville differintegral, as computed by `differint()` (page 899):

```
>>> mp.dps = 15
>>> besselj(1, 3.5, 0.75)
-0.385977722939384
>>> differint(lambda x: besselj(1, x), 3.5, 0.75)
-0.385977722939384
```

mpmath.**j0**(*x*)
Computes the Bessel function $J_0(x)$. See besselj() (page 715).

mpmath.**j1**(*x*)
Computes the Bessel function $J_1(x)$. See besselj() (page 715).

bessely()
mpmath.**bessely**(*n, x, derivative=0*)
bessely(n, x, derivative=0) gives the Bessel function of the second kind,

$$Y_n(x) = \frac{J_n(x)\cos(\pi n) - J_{-n}(x)}{\sin(\pi n)}.$$

For n an integer, this formula should be understood as a limit. With *derivative* = $m \neq 0$, the m-th derivative

$$\frac{d^m}{dx^m} Y_n(x)$$

is computed.

Plots

```
# Bessel function of 2nd kind Y_n(x) on the real line for n=0,1,2,3
y0 = lambda x: bessely(0,x)
y1 = lambda x: bessely(1,x)
y2 = lambda x: bessely(2,x)
y3 = lambda x: bessely(3,x)
plot([y0,y1,y2,y3],[0,10],[-4,1])
```

```
# Bessel function of 2nd kind Y_n(z) in the complex plane
cplot(lambda z: bessely(1,z), [-8,8], [-8,8], points=50000)
```

Examples

Some values of $Y_n(x)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> bessely(0,0), bessely(1,0), bessely(2,0)
(-inf, -inf, -inf)
>>> bessely(1, pi)
0.3588729167767189594679827
>>> bessely(0.5, 3+4j)
(9.242861436961450520325216 - 3.085042824915332562522402j)
```

Arguments may be large:

```
>>> bessely(0, 10000)
0.00364780555898660588668872
>>> bessely(2.5, 10**50)
-4.895250041205098929577 4e-26
>>> bessely(2.5, -10**50)
(0.0 + 4.895250041205098929577 4e-26j)
```

Derivatives and antiderivatives of any order can be computed:

```
>>> bessely(2, 3.5, 1)
0.384261882042266006608923 1
```

```
>>> diff(lambda x: bessely(2, x), 3.5)
0.384261882042266006608923 1
>>> bessely(0.5, 3.5, 1)
-0.206659830415676433790041 7
>>> diff(lambda x: bessely(0.5, x), 3.5)
-0.206659830415676433790041 7
>>> diff(lambda x: bessely(2, x), 0.5, 10)
-208173867409.5547350101511
>>> bessely(2, 0.5, 10)
-208173867409.5547350101511
>>> bessely(2, 100.5, 100)
0.026684875473013723348490 43
>>> quad(lambda x: bessely(2,x), [1,3])
-1.377046859093181969213262
>>> bessely(2,3,-1) - bessely(2,1,-1)
-1.377046859093181969213262
```

besseli()

mpmath.**besseli**(*n, x, derivative=0*)

besseli(n, x, derivative=0) gives the modified Bessel function of the first kind,

$$I_n(x) = i^{-n} J_n(ix).$$

With *derivative* = $m \neq 0$, the m-th derivative

$$\frac{d^m}{dx^m} I_n(x)$$

is computed.

Plots

```
# Modified Bessel function I_n(x) on the real line for n=0,1,2,3
i0 = lambda x: besseli(0,x)
i1 = lambda x: besseli(1,x)
i2 = lambda x: besseli(2,x)
i3 = lambda x: besseli(3,x)
plot([i0,i1,i2,i3],[0,5],[0,5])
```

```
# Modified Bessel function I_n(z) in the complex plane
cplot(lambda z: besseli(1,z), [-8,8], [-8,8], points=50000)
```

Examples

Some values of $I_n(x)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besseli(0,0)
1.0
>>> besseli(1,0)
0.0
>>> besseli(0,1)
1.2660658777520083355982 45
>>> besseli(3.5, 2+3j)
(-0.29043697526425381442890 25 - 0.44690983976548158373070 06j)
```

Arguments may be large:

```
>>> besseli(2, 1000)
2.4807172101918524406167 82e+432
>>> besseli(2, 10**10)
4.2996028516240279003353 91e+4342944813
>>> besseli(2, 6000+10000j)
(-2.1146507532395808271442 04e+2603 + 4.3850402212416290413518 86e+2602j)
```

For integers n, the following integral representation holds:

```
>>> mp.dps = 15
>>> n = 3
>>> x = 2.3
```

```
>>> quad(lambda t: exp(x*cos(t))*cos(n*t), [0,pi])/pi
0.349223221159309
>>> besseli(n,x)
0.349223221159309
```

Derivatives and antiderivatives of any order can be computed:

```
>>> mp.dps = 25
>>> besseli(2, 7.5, 1)
195.82290389313990625658 83
>>> diff(lambda x: besseli(2,x), 7.5)
195.82290389313990625658 83
>>> besseli(2, 7.5, 10)
153.3296508971734525525176
>>> diff(lambda x: besseli(2,x), 7.5, 10)
153.3296508971734525525176
>>> besseli(2,7.5,-1) - besseli(2,3.5,-1)
202.5043900051930141956876
>>> quad(lambda x: besseli(2,x), [3.5, 7.5])
202.5043900051930141956876
```

besselk()

mpmath.**besselk**(*n*, *x*)

`besselk(n, x)` gives the modified Bessel function of the second kind,

$$K_n(x) = \frac{\pi}{2} \frac{I_{-n}(x) - I_n(x)}{\sin(\pi n)}$$

For n an integer, this formula should be understood as a limit.

Plots

```
# Modified Bessel function of 2nd kind K_n(x) on the real line for n=0,1,2,3
k0 = lambda x: besselk(0,x)
k1 = lambda x: besselk(1,x)
k2 = lambda x: besselk(2,x)
k3 = lambda x: besselk(3,x)
plot([k0,k1,k2,k3],[0,8],[0,5])
```

```
# Modified Bessel function of 2nd kind K_n(z) in the complex plane
cplot(lambda z: besselk(1,z), [-8,8], [-8,8], points=50000)
```

Examples

Evaluation is supported for arbitrary complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besselk(0,1)
0.4210244382407083333356274
>>> besselk(0, -1)
(0.4210244382407083333356274 - 3.977463260506422637256 61j)
>>> besselk(3.5, 2+3j)
(-0.02090732889633760668464128 + 0.2464022641351420167819697j)
>>> besselk(2+3j, 0.5)
(0.9615816021726349402626083 + 0.1918250181801757416908224j)
```

Arguments may be large:

```
>>> besselk(0, 100)
4.656628229175902018939005e-45
>>> besselk(1, 10**6)
4.131967049321725588398296e-434298
>>> besselk(1, 10**6*j)
(0.001140348428252385844876706 - 0.0005200017201681152909000961j)
>>> besselk(4.5, fmul(10**50, j, exact=True))
(1.561034538142413947789221e-26 + 1.243554598118700063281496e-25j)
```

The point $x = 0$ is a singularity (logarithmic if $n = 0$):

```
>>> besselk(0,0)
+inf
>>> besselk(1,0)
+inf
>>> for n in range(-4, 5):
...     print(besselk(n, '1e-1000'))
...
4.8e+4001
8.0e+3000
2.0e+2000
1.0e+1000
2302.701024509704096466802
1.0e+1000
2.0e+2000
8.0e+3000
4.8e+4001
```

Bessel function zeros

besseljzero()

mpmath.**besseljzero**(*v, m, derivative=0*)

For a real order $\nu \geq 0$ and a positive integer m, returns $j_{\nu,m}$, the m-th positive zero of the Bessel function of the first kind $J_\nu(z)$ (see besselj() (page 715)). Alternatively, with *derivative=1*, gives the first nonnegative simple zero $j'_{\nu,m}$ of $J'_\nu(z)$.

The indexing convention is that used by Abramowitz & Stegun and the DLMF. Note the special case $j'_{0,1} = 0$, while all other zeros are positive. In effect, only simple zeros are counted (all zeros of Bessel functions are simple except possibly $z = 0$) and $j_{\nu,m}$ becomes a monotonic function of both ν and m.

The zeros are interlaced according to the inequalities

$$j'_{\nu,k} < j_{\nu,k} < j'_{\nu,k+1}$$
$$j_{\nu,1} < j_{\nu+1,2} < j_{\nu,2} < j_{\nu+1,2} < j_{\nu,3} < \cdots$$

Examples

Initial zeros of the Bessel functions $J_0(z), J_1(z), J_2(z)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besseljzero(0,1); besseljzero(0,2); besseljzero(0,3)
2.404825557695772768621632
5.520078110286310649596604
8.653727912911012216954199
>>> besseljzero(1,1); besseljzero(1,2); besseljzero(1,3)
3.831705970207512315614436
7.015586669815618753537 05
10.17346813506272207718571
>>> besseljzero(2,1); besseljzero(2,2); besseljzero(2,3)
5.135622301840682556301402
8.417244140399864857783614
11.61984117214905942709415
```

Initial zeros of $J'_0(z), J'_1(z), J'_2(z)$:

```
0.0
3.831705970207512315614436
7.015586669815618753537 05
>>> besseljzero(1,1,1); besseljzero(1,2,1); besseljzero(1,3,1)
1.841183781340659302643 63
5.331442773525032636884016
8.536316366346285834358961
>>> besseljzero(2,1,1); besseljzero(2,2,1); besseljzero(2,3,1)
3.054236928227140322755932
6.706133194158459146634394
9.969467823087595793179143
```

Zeros with large index:

```
>>> besseljzero(0,100); besseljzero(0,1000); besseljzero(0,10000)
313.3742660775278447196902
3140.807295225078628895545
31415.14114171350798533666
>>> besseljzero(5,100); besseljzero(5,1000); besseljzero(5,10000)
321.1893195676003157339222
3148.657306813047523500494
31422.9947255486291798943
>>> besseljzero(0,100,1); besseljzero(0,1000,1); besseljzero(0,10000,1)
311.8018681873704508125112
3139.236339643802482833973
31413.57032947022399485808
```

Zeros of functions with large order:

```
>>> besseljzero(50,1)
57.11689916011917411936228
>>> besseljzero(50,2)
62.80769876483536093435393
>>> besseljzero(50,100)
388.6936600656058834640981
>>> besseljzero(50,1,1)
52.99764038731665010944037
>>> besseljzero(50,2,1)
60.02631933279942589882363
>>> besseljzero(50,100,1)
387.1083151608726181086283
```

Zeros of functions with fractional order:

```
>>> besseljzero(0.5,1); besseljzero(1.5,1); besseljzero(2.25,4)
3.141592653589793238462643
4.493409457909064175307881
15.15657692957458622921634
```

Both $J_\nu(z)$ and $J'_\nu(z)$ can be expressed as infinite products over their zeros:

```
>>> v,z = 2, mpf(1)
>>> (z/2)**v/gamma(v+1) * \
...     nprod(lambda k: 1-(z/besseljzero(v,k))**2, [1,inf])
...
0.11490348493190048046964 69
>>> besselj(v,z)
0.11490348493190048046964 69
>>> (z/2)**(v-1)/2/gamma(v) * \
...     nprod(lambda k: 1-(z/besseljzero(v,k,1))**2, [1,inf])
```

```
...
0.210243615881132555020388 4
>>> besselj(v,z,1)
0.210243615881132555020388 4
```

besselyzero()

mpmath.**besselyzero**(*v, m, derivative=0*)

For a real order $\nu \geq 0$ and a positive integer m, returns $y_{\nu,m}$, the m-th positive zero of the Bessel function of the second kind $Y_\nu(z)$ (see `bessely()` (page 719)). Alternatively, with *derivative=1*, gives the first positive zero $y'_{\nu,m}$ of $Y'_\nu(z)$.

The zeros are interlaced according to the inequalities

$$y_{\nu,k} < y'_{\nu,k} < y_{\nu,k+1}$$

$$y_{\nu,1} < y_{\nu+1,2} < y_{\nu,2} < y_{\nu+1,2} < y_{\nu,3} < \cdots$$

Examples

Initial zeros of the Bessel functions $Y_0(z), Y_1(z), Y_2(z)$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> besselyzero(0,1); besselyzero(0,2); besselyzero(0,3)
0.8935769662791675215848871
3.957678419314857868375677
7.086051060301772697623625
>>> besselyzero(1,1); besselyzero(1,2); besselyzero(1,3)
2.197141326031017035149034
5.429681040794135132772005
8.596005868331168926429606
>>> besselyzero(2,1); besselyzero(2,2); besselyzero(2,3)
3.384241767149593472701426
6.793807513268267538291167
10.02347797936003797850539
```

Initial zeros of $Y'_0(z), Y'_1(z), Y'_2(z)$:

```
>>> besselyzero(0,1,1); besselyzero(0,2,1); besselyzero(0,3,1)
2.197141326031017035149034
5.429681040794135132772005
8.596005868331168926429606
>>> besselyzero(1,1,1); besselyzero(1,2,1); besselyzero(1,3,1)
3.683022856585177699898967
6.941499953654175655751944
10.12340465543661307978775
>>> besselyzero(2,1,1); besselyzero(2,2,1); besselyzero(2,3,1)
5.002582931446063945200176
8.350724701413079526349714
11.57419546521764654624265
```

Zeros with large index:

```
>>> besselyzero(0,100); besselyzero(0,1000); besselyzero(0,10000)
311.8034717601871549333419
3139.236498918198006794026
31413.57034538691205229188
>>> besselyzero(5,100); besselyzero(5,1000); besselyzero(5,10000)
319.6183338562782156235062
```

```
3147.08650852455640447318 6
31421.42392920214673402828
>>> besselyzero(0,100,1); besselyzero(0,1000,1); besselyzero(0,10000,1)
313.37267054263593450504 49
3140.80713603034021361006 5
31415.14112575976157822017 5
```

Zeros of functions with large order:

```
>>> besselyzero(50,1)
53.50285882040036394680237
>>> besselyzero(50,2)
60.11244442774058114686022
>>> besselyzero(50,100)
387.10965098249439577068 35
>>> besselyzero(50,1,1)
56.96290427516751320063605
>>> besselyzero(50,2,1)
62.74888166945933944036623
>>> besselyzero(50,100,1)
388.69233005483092583554 75
```

Zeros of functions with fractional order:

```
>>> besselyzero(0.5,1); besselyzero(1.5,1); besselyzero(2.25,4)
1.5707963267948966192313 22
2.7983860457838871367202 49
13.567212087707351233760 18
```

Hankel functions

hankel1()

mpmath.**hankel1**(*n*, *x*)

`hankel1(n,x)` computes the Hankel function of the first kind, which is the complex combination of Bessel functions given by

$$H_n^{(1)}(x) = J_n(x) + iY_n(x).$$

Plots

```
# Hankel function H1_n(x) on the real line for n=0,1,2,3
h0 = lambda x: hankel1(0,x)
h1 = lambda x: hankel1(1,x)
h2 = lambda x: hankel1(2,x)
h3 = lambda x: hankel1(3,x)
plot([h0,h1,h2,h3],[0,6],[-2,1])
```

```
# Hankel function H1_n(z) in the complex plane
cplot(lambda z: hankel1(1,z), [-8,8], [-8,8], points=50000)
```

Examples

The Hankel function is generally complex-valued:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hankel1(2, pi)
(0.4854339326315091097054957 - 0.09990071392902787877349 03j)
>>> hankel1(3.5, pi)
(0.2340002029630507922628888 - 0.6419643823412927142424049j)
```

hankel2()

mpmath.**hankel2**(n, x)

hankel2(n,x) computes the Hankel function of the second kind, which is the complex combination of Bessel functions given by

$$H_n^{(2)}(x) = J_n(x) - iY_n(x).$$

Plots

```
# Hankel function H2_n(x) on the real line for n=0,1,2,3
h0 = lambda x: hankel2(0,x)
h1 = lambda x: hankel2(1,x)
h2 = lambda x: hankel2(2,x)
h3 = lambda x: hankel2(3,x)
plot([h0,h1,h2,h3],[0,6],[-1,2])
```

```
# Hankel function H2_n(z) in the complex plane
cplot(lambda z: hankel2(1,z), [-8,8], [-8,8], points=50000)
```

Examples

The Hankel function is generally complex-valued:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hankel2(2, pi)
(0.4854339326315091097054957 + 0.09990071392902787877349 03j)
>>> hankel2(3.5, pi)
(0.2340002029630507922628888 + 0.6419643823412927142424049j)
```

Kelvin functions

ber()

mpmath.**ber**(*ctx*, *n*, *z*, ***kwargs*)

Computes the Kelvin function ber, which for real arguments gives the real part of the Bessel J function of a rotated argument

$$J_n\left(xe^{3\pi i/4}\right) = \mathrm{ber}_n(x) + i\mathrm{bei}_n(x).$$

The imaginary part is given by `bei()` (page 736).

Plots

```
# Kelvin functions ber_n(x) and bei_n(x) on the real line for n=0,2
f0 = lambda x: ber(0,x)
f1 = lambda x: bei(0,x)
f2 = lambda x: ber(2,x)
f3 = lambda x: bei(2,x)
plot([f0,f1,f2,f3],[0,10],[-10,10])
```

Examples

Verifying the defining relation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> n, x = 2, 3.5
>>> ber(n,x)
1.442338852571888752631129
>>> bei(n,x)
-0.948359035324558320217678
>>> besselj(n, x*root(1,8,3))
(1.442338852571888752631129 - 0.948359035324558320217678j)
```

The ber and bei functions are also defined by analytic continuation for complex arguments:

```
>>> ber(1+j, 2+3j)
(4.675445984756614424069563 - 15.84901771719130765656316j)
>>> bei(1+j, 2+3j)
(15.83886679193707699364398 + 4.684053288183046528703611j)
```

bei()

mpmath.**bei**(*ctx*, *n*, *z*, ***kwargs*)

Computes the Kelvin function bei, which for real arguments gives the imaginary part of the Bessel J function of a rotated argument. See ber() (page 734).

ker()

mpmath.**ker**(*ctx*, *n*, *z*, ***kwargs*)

Computes the Kelvin function ker, which for real arguments gives the real part of the (rescaled) Bessel K function of a rotated argument

$$e^{-\pi i/2} K_n\left(x e^{3\pi i/4}\right) = \mathrm{ker}_n(x) + i\mathrm{kei}_n(x).$$

The imaginary part is given by kei() (page 737).

Plots

```
# Kelvin functions ker_n(x) and kei_n(x) on the real line for n=0,2
f0 = lambda x: ker(0,x)
f1 = lambda x: kei(0,x)
f2 = lambda x: ker(2,x)
f3 = lambda x: kei(2,x)
plot([f0,f1,f2,f3],[0,5],[-1,4])
```

Examples

Verifying the defining relation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> n, x = 2, 4.5
>>> ker(n,x)
0.02542895201906369640249801
>>> kei(n,x)
-0.02074960467222823237055351
>>> exp(-n*pi*j/2) * besselk(n, x*root(1,8,1))
(0.02542895201906369640249801 - 0.02074960467222823237055351j)
```

The ker and kei functions are also defined by analytic continuation for complex arguments:

```
>>> ker(1+j, 3+4j)
(1.586084268115490421090533 - 2.939717517906339193598719j)
>>> kei(1+j, 3+4j)
(-2.940403256319453402690132 - 1.585621643835618941044855j)
```

kei()

mpmath.**kei**(*ctx, n, z, **kwargs*)

Computes the Kelvin function kei, which for real arguments gives the imaginary part of the (rescaled) Bessel K function of a rotated argument. See `ker()` (page 736).

Struve functions

struveh()

mpmath.**struveh**(*ctx, n, z, **kwargs*)

Gives the Struve function

$$\mathbf{H}_n(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma(k+\frac{3}{2})\Gamma(k+n+\frac{3}{2})} \left(\frac{z}{2}\right)^{2k+n+1}$$

which is a solution to the Struve differential equation

$$z^2 f''(z) + z f'(z) + (z^2 - n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!}.$$

Examples

Evaluation for arbitrary real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> struveh(0, 3.5)
0.3608207733778295024977797
>>> struveh(-1, 10)
-0.2552127197269567680347 32
>>> struveh(1, -100.5)
0.5819566816797362287502246
>>> struveh(2.5, 10000000000000)
3153915652525200060.308937
>>> struveh(2.5, -10000000000000)
(0.0 - 3153915652525200060.308937j)
>>> struveh(1+j, 1000000+4000000j)
(-3.066421087689197632388731e+1737173 - 1.596619701076529803290973e+1737173j)
```

A Struve function of half-integer order is elementary; for example:

```
>>> z = 3
>>> struveh(0.5, 3)
0.916707686756413817867159 5
>>> sqrt(2/(pi*z))*(1-cos(z))
0.916707686756413817867159 5
```

Numerically verifying the differential equation:

```
>>> z = mpf(4.5)
>>> n = 3
>>> f = lambda z: struveh(n,z)
>>> lhs = z**2*diff(f,z,2) + z*diff(f,z) + (z**2-n**2)*f(z)
>>> rhs = 2*z**(n+1)/fac2(2*n-1)/pi
>>> lhs
17.40359302709875496632744
>>> rhs
17.40359302709875496632744
```

struvel()

mpmath.**struvel**(*ctx, n, z, **kwargs*)

Gives the modified Struve function

$$\mathbf{L}_n(z) = -ie^{-n\pi i/2}\mathbf{H}_n(iz)$$

which solves to the modified Struve differential equation

$$z^2 f''(z) + zf'(z) - (z^2 + n^2)f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!}.$$

Examples

Evaluation for arbitrary real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> struvel(0, 3.5)
7.180846515103737996249972
>>> struvel(-1, 10)
2670.994904980850550721511
>>> struvel(1, -100.5)
1.757089288053346261497686e+42
>>> struvel(2.5, 10000000000000)
4.160893281017115450519948e+4342944819025
>>> struvel(2.5, -10000000000000)
(0.0 - 4.160893281017115450519948e+4342944819025j)
>>> struvel(1+j, 700j)
(-0.172115004948007945124607 6 + 0.124077095312683109346405 5j)
>>> struvel(1+j, 1000000+4000000j)
(-2.973341637511505389128708e+434290 - 5.164633059729968297147448e+434290j)
```

Numerically verifying the differential equation:

```
>>> z = mpf(3.5)
>>> n = 3
>>> f = lambda z: struvel(n,z)
>>> lhs = z**2*diff(f,z,2) + z*diff(f,z) - (z**2+n**2)*f(z)
>>> rhs = 2*z**(n+1)/fac2(2*n-1)/pi
```

```
>>> lhs
6.36885030600600678353018165
>>> rhs
6.36885030600600678353018165
```

Anger-Weber functions

angerj()

mpmath.**angerj**(*ctx, v, z, **kwargs*)

Gives the Anger function

$$\mathbf{J}_\nu(z) = \frac{1}{\pi}\int_0^\pi \cos(\nu t - z\sin t)dt$$

which is an entire function of both the parameter ν and the argument z. It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z}f'(z) + \left(1 - \frac{\nu^2}{z^2}\right)f(z) = \frac{(z-\nu)}{\pi z^2}\sin(\pi\nu).$$

Examples

Evaluation for real and complex parameter and argument:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> angerj(2,3)
0.486091260585891076907831 1
>>> angerj(-3+4j, 2+5j)
(-5033.358320403384472395612 + 585.8011892476145118551756j)
>>> angerj(3.25, 1e6j)
(4.63074363971589334657074 3e+434290 - 1.11796040988750590684845 6e+434291j)
>>> angerj(-1.5, 1e6)
0.00027957197470738793930870 11
```

The Anger function coincides with the Bessel J-function when ν is an integer:

```
>>> angerj(1,3); besselj(1,3)
0.33905895852593645892551 46
0.33905895852593645892551 46
>>> angerj(1.5,3); besselj(1.5,3)
0.40889698486910808593288 47
0.47771821508709177155150 15
```

Verifying the differential equation:

```
>>> v,z = mpf(2.25), 0.75
>>> f = lambda z: angerj(v,z)
>>> diff(f,z,2) + diff(f,z)/z + (1-(v/z)**2)*f(z)
-0.600210877438070713036799 5
>>> (z-v)/(pi*z**2) * sinpi(v)
-0.600210877438070713036799 5
```

Verifying the integral representation:

```
>>> angerj(v,z)
0.114538075991933318090050 1
>>> quad(lambda t: cos(v*t-z*sin(t))/pi, [0,pi])
0.114538075991933318090050 1
```

References

1.[DLMF] (page 1625) section 11.10: Anger-Weber Functions

webere()

mpmath.**webere**(*ctx, v, z, **kwargs*)

Gives the Weber function

$$\mathbf{E}_\nu(z) = \frac{1}{\pi}\int_0^\pi \sin(\nu t - z\sin t)dt$$

which is an entire function of both the parameter ν and the argument z. It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z}f'(z) + \left(1 - \frac{\nu^2}{z^2}\right)f(z) = -\frac{1}{\pi z^2}(z + \nu + (z-\nu)\cos(\pi\nu)).$$

Examples

Evaluation for real and complex parameter and argument:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> webere(2,3)
-0.1057668973099018425662646
>>> webere(-3+4j, 2+5j)
(-585.8081418209852019290498 - 5033.314488899926921597203j)
>>> webere(3.25, 1e6j)
(-1.117960409887505906848456e+434291 - 4.630743639715893346570743e+434290j)
>>> webere(3.25, 1e6)
-0.00002812518265894315604914453
```

Up to addition of a rational function of z, the Weber function coincides with the Struve H-function when ν is an integer:

```
>>> webere(1,3); 2/pi-struveh(1,3)
-0.383489796818869017737288 1
-0.3834897968188690177372881
>>> webere(5,3); 26/(35*pi)-struveh(5,3)
0.2009680659308154011878075
0.2009680659308154011878075
```

Verifying the differential equation:

```
>>> v,z = mpf(2.25), 0.75
>>> f = lambda z: webere(v,z)
>>> diff(f,z,2) + diff(f,z)/z + (1-(v/z)**2)*f(z)
-1.097441848875479535164627
>>> -(z+v+(z-v)*cospi(v))/(pi*z**2)
-1.097441848875479535164627
```

Verifying the integral representation:

```
>>> webere(v,z)
0.1486507351534283744485421
>>> quad(lambda t: sin(v*t-z*sin(t))/pi, [0,pi])
0.1486507351534283744485421
```

References

1.[DLMF] (page 1625) section 11.10: Anger-Weber Functions

Lommel functions

lommels1()

mpmath.**lommels1**(*ctx, u, v, z, **kwargs*)

Gives the Lommel function $s_{\mu,\nu}$ or $s_{\mu,\nu}^{(1)}$

$$s_{\mu,\nu}(z) = \frac{z^{\mu+1}}{(\mu-\nu+1)(\mu+\nu+1)}\,{}_1F_2\left(1; \frac{\mu-\nu+3}{2}, \frac{\mu+\nu+3}{2}; -\frac{z^2}{4}\right)$$

which solves the inhomogeneous Bessel equation

$$z^2 f''(z) + z f'(z) + (z^2 - \nu^2) f(z) = z^{\mu+1}.$$

A second solution is given by `lommels2()` (page 742).

Plots

```
# Lommel function s_(u,v)(x) on the real line for a few different u,v
f1 = lambda x: lommels1(-1,2.5,x)
f2 = lambda x: lommels1(0,0.5,x)
f3 = lambda x: lommels1(0,6,x)
f4 = lambda x: lommels1(0.5,3,x)
plot([f1,f2,f3,f4], [0,20])
```

Examples

An integral representation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> u,v,z = 0.25, 0.125, mpf(0.75)
>>> lommels1(u,v,z)
0.4276243877565150372999126
>>> (bessely(v,z)*quad(lambda t: t**u*besselj(v,t), [0,z]) - \
...  besselj(v,z)*quad(lambda t: t**u*bessely(v,t), [0,z]))*(pi/2)
0.4276243877565150372999126
```

A special value:

```
>>> lommels1(v,v,z)
0.5461221367746048054932553
>>> gamma(v+0.5)*sqrt(pi)*power(2,v-1)*struveh(v,z)
0.5461221367746048054932553
```

Verifying the differential equation:

```
>>> f = lambda z: lommels1(u,v,z)
>>> z**2*diff(f,z,2) + z*diff(f,z) + (z**2-v**2)*f(z)
0.6979536443265746992059141
>>> z**(u+1)
0.6979536443265746992059141
```

References

1. [GradshteynRyzhik] (page 1625)
2. [Weisstein] (page 1626) http://mathworld.wolfram.com/LommelFunction.html

lommels2()

mpmath.**lommels2**(*ctx, u, v, z, **kwargs*)

Gives the second Lommel function $S_{\mu,\nu}$ or $s_{\mu,\nu}^{(2)}$

$$S_{\mu,\nu}(z) = s_{\mu,\nu}(z) + 2^{\mu-1}\Gamma\left(\tfrac{1}{2}(\mu-\nu+1)\right)\Gamma\left(\tfrac{1}{2}(\mu+\nu+1)\right)\times \left[\sin(\tfrac{1}{2}(\mu-\nu)\pi)J_\nu(z) - \cos(\tfrac{1}{2}(\mu-\nu)\pi)Y_\nu(z)\right]$$

which solves the same differential equation as `lommels1()` (page 741).

Plots

```
# Lommel function S_(u,v)(x) on the real line for a few different u,v
f1 = lambda x: lommels2(-1,2.5,x)
f2 = lambda x: lommels2(1.5,2,x)
f3 = lambda x: lommels2(2.5,1,x)
f4 = lambda x: lommels2(3.5,-0.5,x)
plot([f1,f2,f3,f4], [0,8], [-8,8])
```

Examples

For large $|z|$, $S_{\mu,\nu} \sim z^{\mu-1}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> lommels2(10,2,30000)
1.968299831601008419949804e+40
>>> power(30000,9)
1.9683e+40
```

A special value:

```
>>> u,v,z = 0.5, 0.125, mpf(0.75)
>>> lommels2(v,v,z)
0.9589683199624672099969765
>>> (struveh(v,z)-bessely(v,z))*power(2,v-1)*sqrt(pi)*gamma(v+0.5)
0.9589683199624672099969765
```

Verifying the differential equation:

```
>>> f = lambda z: lommels2(u,v,z)
>>> z**2*diff(f,z,2) + z*diff(f,z) + (z**2-v**2)*f(z)
0.6495190528383289850727924
>>> z**(u+1)
0.6495190528383289850727924
```

References

1. [GradshteynRyzhik] (page 1625)

2.[Weisstein] (page 1626) http://mathworld.wolfram.com/LommelFunction.html

Airy and Scorer functions

airyai()

mpmath.**airyai**(*z*, *derivative=0*, ***kwargs*)

Computes the Airy function $\mathrm{Ai}(z)$, which is the solution of the Airy differential equation $f''(z) - zf(z) = 0$ with initial conditions

$$\mathrm{Ai}(0) = \frac{1}{3^{2/3}\Gamma\left(\frac{2}{3}\right)}$$
$$\mathrm{Ai}'(0) = -\frac{1}{3^{1/3}\Gamma\left(\frac{1}{3}\right)}.$$

Other common ways of defining the Ai-function include integrals such as

$$\mathrm{Ai}(x) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt \qquad x \in \mathbb{R}$$
$$\mathrm{Ai}(z) = \frac{\sqrt{3}}{2\pi}\int_0^\infty \exp\left(-\frac{t^3}{3} - \frac{z^3}{3t^3}\right) dt.$$

The Ai-function is an entire function with a turning point, behaving roughly like a slowly decaying sine wave for $z < 0$ and like a rapidly decreasing exponential for $z > 0$. A second solution of the Airy differential equation is given by $\mathrm{Bi}(z)$ (see `airybi()` (page 749)).

Optionally, with *derivative=alpha*, `airyai()` can compute the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Ai}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral

$$f_0(z) = \mathrm{Ai}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt.$$

The Ai-function has infinitely many zeros, all located along the negative half of the real axis. They can be computed with `airyaizero()` (page 753).

Plots

```
# Airy function Ai(x), Ai'(x) and int_0^x Ai(t) dt on the real line
f = airyai
f_diff = lambda z: airyai(z, derivative=1)
f_int = lambda z: airyai(z, derivative=-1)
plot([f, f_diff, f_int], [-10,5])
```

```
# Airy function Ai(z) in the complex plane
cplot(airyai, [-8,8], [-8,8], points=50000)
```

Basic examples

Limits and values include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airyai(0); 1/(power(3,'2/3')*gamma('2/3'))
0.3550280538878172392600632
0.3550280538878172392600632
>>> airyai(1)
0.1352924163128814155241474
>>> airyai(-1)
0.5355608832923521187995166
>>> airyai(inf); airyai(-inf)
0.0
0.0
```

Evaluation is supported for large magnitudes of the argument:

```
>>> airyai(-100)
0.1767533932395528780908311
>>> airyai(100)
2.634482152088184489550553e-291
>>> airyai(50+50j)
(-5.317901957074564040998 17e-68 - 1.163588003770709748720107e-67j)
>>> airyai(-50+50j)
(1.041242537363167632587245e+158 + 3.347525544923600321838281e+157j)
```

Huge arguments are also fine:

```
>>> airyai(10**10)
1.1622359782987417799536 93e-289529654602171
>>> airyai(-10**10)
0.00017362064481528185105 10181
>>> w = airyai(10**10*(1+j))
>>> w.real
5.7115086837213555283225 67e-186339621747698
>>> w.imag
1.8672455069623125778481 66e-186339621747697
```

The first root of the Ai-function is:

```
>>> findroot(airyai, -2)
-2.3381074104597670384891 97
>>> airyaizero(1)
-2.3381074104597670384891 97
```

Properties and relations

Verifying the Airy differential equation:

```
>>> for z in [-3.4, 0, 2.5, 1+2j]:
...     chop(airyai(z,2) - z*airyai(z))
...
0.0
0.0
0.0
0.0
```

The first few terms of the Taylor series expansion around $z = 0$ (every third term is zero):

```
>>> nprint(taylor(airyai, 0, 5))
[0.355028, -0.258819, 0.0, 0.0591713, -0.0215683, 0.0]
```

The Airy functions satisfy the Wronskian relation $\mathrm{Ai}(z)\,\mathrm{Bi}'(z) - \mathrm{Ai}'(z)\,\mathrm{Bi}(z) = 1/\pi$:

```
>>> z = -0.5
>>> airyai(z)*airybi(z,1) - airyai(z,1)*airybi(z)
0.31830988618379067153776 75
>>> 1/pi
0.31830988618379067153776 75
```

The Airy functions can be expressed in terms of Bessel functions of order $\pm 1/3$. For $\Re[z] \le 0$, we have:

```
>>> z = -3
>>> airyai(z)
-0.37881429367765807434724 39
>>> y = 2*power(-z,'3/2')/3
>>> (sqrt(-z) * (besselj('1/3',y) + besselj('-1/3',y)))/3
-0.37881429367765807434724 39
```

Derivatives and integrals

Derivatives of the Ai-function (directly and using `diff()` (page 896)):

```
>>> airyai(-3,1); diff(airyai,-3)
0.31458376921659881365078 73
0.31458376921659881365078 73
>>> airyai(-3,2); diff(airyai,-3,2)
1.1364428810329742230417 32
```

```
1.136442881032974223041732
>>> airyai(1000,1); diff(airyai,1000)
-2.9431339179103360904597 48e-9156
-2.9431339179103360904597 48e-9156
```

Several derivatives at $z = 0$:

```
>>> airyai(0,0); airyai(0,1); airyai(0,2)
0.355028053887817239260063 2
-0.258819403792806798405183 6
0.0
>>> airyai(0,3); airyai(0,4); airyai(0,5)
0.355028053887817239260063 2
-0.517638807585613596810367 1
0.0
>>> airyai(0,15); airyai(0,16); airyai(0,17)
1292.30211615165475090663
-3188.65505472737975635186 1
0.0
```

The integral of the Ai-function:

```
>>> airyai(3,-1); quad(airyai, [0,3])
0.329920376007021772500270 1
0.329920376007021772500270 1
>>> airyai(-10,-1); quad(airyai, [0,-10])
-0.765698403134212917425148
-0.765698403134212917425148
```

Integrals of high or fractional order:

```
>>> airyai(-2,0.5); differint(airyai,-2,0.5,0)
(0.0 + 0.245359610135143827384472 5j)
(0.0 + 0.245359610135143827384472 5j)
>>> airyai(-2,-4); differint(airyai,-2,-4,0)
0.293917644163680958033936 5
0.293917644163680958033936 5
>>> airyai(0,-1); airyai(0,-2); airyai(0,-3)
0.0
0.0
0.0
```

Integrals of the Ai-function can be evaluated at limit points:

```
>>> airyai(-1000000,-1); airyai(-inf,-1)
-0.666684372831153997875151 2
-0.666666666666666666666667
>>> airyai(10,-1); airyai(+inf,-1)
0.333333333299169015942793 2
0.333333333333333333333333
>>> airyai(+inf,-2); airyai(+inf,-3)
+inf
+inf
>>> airyai(-1000000,-2); airyai(-inf,-2)
666666.407847265065120974 2
+inf
>>> airyai(-1000000,-3); airyai(-inf,-3)
-333333074513.752026499573 3
-inf
```

References

1. [DLMF] (page 1625) Chapter 9: Airy and Related Functions
2. [WolframFunctions] (page 1626) section: Bessel-Type Functions

airybi()

mpmath.**airybi**(*z*, *derivative=0*, ***kwargs*)

Computes the Airy function $\mathrm{Bi}(z)$, which is the solution of the Airy differential equation $f''(z) - zf(z) = 0$ with initial conditions

$$\mathrm{Bi}(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{2}{3}\right)}$$
$$\mathrm{Bi}'(0) = \frac{3^{1/6}}{\Gamma\left(\frac{1}{3}\right)}.$$

Like the Ai-function (see airyai() (page 744)), the Bi-function is oscillatory for $z < 0$, but it grows rather than decreases for $z > 0$.

Optionally, as for airyai() (page 744), derivatives, integrals and fractional derivatives can be computed with the *derivative* parameter.

The Bi-function has infinitely many zeros along the negative half-axis, as well as complex zeros, which can all be computed with airybizero() (page 754).

Plots

```
# Airy function Bi(x), Bi'(x) and int_0^x Bi(t) dt on the real line
f = airybi
f_diff = lambda z: airybi(z, derivative=1)
f_int = lambda z: airybi(z, derivative=-1)
plot([f, f_diff, f_int], [-10,2], [-1,2])
```

```
# Airy function Bi(z) in the complex plane
cplot(airybi, [-8,8], [-8,8], points=50000)
```

Basic examples

Limits and values include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airybi(0); 1/(power(3,'1/6')*gamma('2/3'))
0.614926627446000735150922 4
0.614926627446000735150922 4
>>> airybi(1)
1.207423594952871259436379
>>> airybi(-1)
0.1039973894969446118886 9
>>> airybi(inf); airybi(-inf)
+inf
0.0
```

Evaluation is supported for large magnitudes of the argument:

```
>>> airybi(-100)
0.0242738876801601316056674 7
>>> airybi(100)
6.04122399667020139900526 5e+288
>>> airybi(50+50j)
(-5.32207626732143566929033 4e+63 + 1.47845029116524378974942 7e+65j)
>>> airybi(-50+50j)
(-3.34752554492360032183828 1e+157 + 1.04124253736316763258724 5e+158j)
```

Huge arguments:

```
>>> airybi(10**10)
1.36938578794353981868843 3e+289529654602165
>>> airybi(-10**10)
0.00177565614169293274761 0973
>>> w = airybi(10**10*(1+j))
>>> w.real
-6.55995593109619687584 5858e+186339621747689
>>> w.imag
-6.82246272698135718092 9024e+186339621747690
```

The first real root of the Bi-function is:

```
>>> findroot(airybi, -1); airybizero(1)
-1.17371322270912792491998
-1.17371322270912792491998
```

Properties and relations

Verifying the Airy differential equation:

```
>>> for z in [-3.4, 0, 2.5, 1+2j]:
...     chop(airybi(z,2) - z*airybi(z))
...
0.0
0.0
0.0
0.0
```

The first few terms of the Taylor series expansion around $z = 0$ (every third term is zero):

```
>>> nprint(taylor(airybi, 0, 5))
[0.614927, 0.448288, 0.0, 0.102488, 0.0373574, 0.0]
```

The Airy functions can be expressed in terms of Bessel functions of order $\pm 1/3$. For $\Re[z] \leq 0$, we have:

```
>>> z = -3
>>> airybi(z)
-0.198289626374926543220 6449
>>> p = 2*power(-z,'3/2')/3
>>> sqrt(-mpf(z)/3)*(besselj('-1/3',p) - besselj('1/3',p))
-0.198289626374926543220 6449
```

Derivatives and integrals

Derivatives of the Bi-function (directly and using `diff()` (page 896)):

```
>>> airybi(-3,1); diff(airybi,-3)
-0.675611222685258537668032
-0.675611222685258537668032
>>> airybi(-3,2); diff(airybi,-3,2)
0.594868879124779629661934 6
0.594868879124779629661934 6
>>> airybi(1000,1); diff(airybi,1000)
1.71005511462461498926233 5e+9156
1.71005511462461498926233 5e+9156
```

Several derivatives at $z = 0$:

```
>>> airybi(0,0); airybi(0,1); airybi(0,2)
0.614926627446000735150922 4
```

```
0.44828835735382635791482 37
0.0
>>> airybi(0,3); airybi(0,4); airybi(0,5)
0.61492662744600073515092 24
0.89657671470765271582964 74
0.0
>>> airybi(0,15); airybi(0,16); airybi(0,17)
2238.332923903442675949357
5522.912562599140729510628
0.0
```

The integral of the Bi-function:

```
>>> airybi(3,-1); quad(airybi, [0,3])
10.06200303130620056316655
10.06200303130620056316655
>>> airybi(-10,-1); quad(airybi, [0,-10])
-0.015040424806140020451354 83
-0.015040424806140020451354 83
```

Integrals of high or fractional order:

```
>>> airybi(-2,0.5); differint(airybi, -2, 0.5, 0)
(0.0 + 0.501985905534169922345 3257j)
(0.0 + 0.501985905534169922345 3257j)
>>> airybi(-2,-4); differint(airybi,-2,-4,0)
0.280931459992244725213 9092
0.280931459992244725213 9092
>>> airybi(0,-1); airybi(0,-2); airybi(0,-3)
0.0
0.0
0.0
```

Integrals of the Bi-function can be evaluated at limit points:

```
>>> airybi(-1000000,-1); airybi(-inf,-1)
0.00000219126112806343404 7966873
0.0
>>> airybi(10,-1); airybi(+inf,-1)
147809803.1074067161675853
+inf
>>> airybi(+inf,-2); airybi(+inf,-3)
+inf
+inf
>>> airybi(-1000000,-2); airybi(-inf,-2)
0.44828837505999084798510 85
0.44828835735382635791482 37
>>> gamma('2/3')*power(3,'2/3')/(2*pi)
0.44828835735382635791482 37
>>> airybi(-100000,-3); airybi(-inf,-3)
-44828.52827206932872493133
-inf
>>> airybi(-100000,-4); airybi(-inf,-4)
2241411040.437759489540248
+inf
```

airyaizero()

mpmath.**airyaizero**(*k*, *derivative=0*)

Gives the k-th zero of the Airy Ai-function, i.e. the k-th number a_k ordered by magnitude for which $\mathrm{Ai}(a_k) = 0$.

Optionally, with *derivative=1*, the corresponding zero a'_k of the derivative function, i.e. $\mathrm{Ai}'(a'_k) = 0$, is computed.

Examples

Some values of a_k:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airyaizero(1)
-2.338107410459767038489197
>>> airyaizero(2)
-4.087949444130970616636989
>>> airyaizero(3)
-5.520559828095551059129856
>>> airyaizero(1000)
-281.0315196125215528353364
```

Some values of a'_k:

```
>>> airyaizero(1,1)
-1.018792971647471089017325
>>> airyaizero(2,1)
-3.248197582179836537875424
>>> airyaizero(3,1)
-4.820099211178735639400616
>>> airyaizero(1000,1)
-280.9378080358935070607097
```

Verification:

```
>>> chop(airyai(airyaizero(1)))
0.0
>>> chop(airyai(airyaizero(1,1),1))
0.0
```

airybizero()

mpmath.**airybizero**(*k*, *derivative=0*, *complex=0*)

With *complex=False*, gives the k-th real zero of the Airy Bi-function, i.e. the k-th number b_k ordered by magnitude for which $\mathrm{Bi}(b_k) = 0$.

With *complex=True*, gives the k-th complex zero in the upper half plane β_k. Also the conjugate $\overline{\beta_k}$ is a zero.

Optionally, with *derivative=1*, the corresponding zero b'_k or β'_k of the derivative function, i.e. $\mathrm{Bi}'(b'_k) = 0$ or $\mathrm{Bi}'(\beta'_k) = 0$, is computed.

Examples

Some values of b_k:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> airybizero(1)
-1.173713222709127924919 98
>>> airybizero(2)
-3.271093302836352715680228
```

```
>>> airybizero(3)
-4.83073784166201593266770 9
>>> airybizero(1000)
-280.9378112034152401578834
```

Some values of b_k:

```
>>> airybizero(1,1)
-2.294439682614123246622459
>>> airybizero(2,1)
-4.073155089071828215552369
>>> airybizero(3,1)
-5.512395729663599496259593
>>> airybizero(1000,1)
-281.0315164471118527161362
```

Some values of β_k:

```
>>> airybizero(1,complex=True)
(0.977544886731620685946992 7 + 2.141290706038744575749139j)
>>> airybizero(2,complex=True)
(1.896775013895336346627217 + 3.627291764358919410440499j)
>>> airybizero(3,complex=True)
(2.633157739354946595708019 + 4.855468179979844983174628j)
>>> airybizero(1000,complex=True)
(140.4978560578493018899793 + 243.3907724215792121244867j)
```

Some values of β_k':

```
>>> airybizero(1,1,complex=True)
(0.214947074537430567608832 9 + 1.100600143302797880647194j)
>>> airybizero(2,1,complex=True)
(1.458168309223507392028211 + 2.912249367458445419235083j)
>>> airybizero(3,1,complex=True)
(2.273760763013482299792362 + 4.254528549217097862167015j)
>>> airybizero(1000,1,complex=True)
(140.4509972835270559730423 + 243.3096175398562811896208j)
```

Verification:

```
>>> chop(airybi(airybizero(1)))
0.0
>>> chop(airybi(airybizero(1,1),1))
0.0
>>> u = airybizero(1,complex=True)
>>> chop(airybi(u))
0.0
>>> chop(airybi(conj(u)))
0.0
```

The complex zeros (in the upper and lower half-planes respectively) asymptotically approach the rays $z = R\exp(\pm i\pi/3)$:

```
>>> arg(airybizero(1,complex=True))
1.142532510286334022305364
>>> arg(airybizero(1000,complex=True))
1.047271114786212061583917
>>> arg(airybizero(1000000,complex=True))
1.047197624741816183341355
```

```
>>> pi/3
1.0471975511965977461542 14
```

scorergi()

mpmath.**scorergi**(*z*, ***kwargs*)

Evaluates the Scorer function

$$\mathrm{Gi}(z) = \mathrm{Ai}(z) \int_0^z \mathrm{Bi}(t)dt + \mathrm{Bi}(z) \int_z^\infty \mathrm{Ai}(t)dt$$

which gives a particular solution to the inhomogeneous Airy differential equation $f''(z) - zf(z) = 1/\pi$. Another particular solution is given by the Scorer Hi-function (`scorerhi()` (page 758)). The two functions are related as $\mathrm{Gi}(z) + \mathrm{Hi}(z) = \mathrm{Bi}(z)$.

Plots

```
# Scorer function Gi(x) and Gi'(x) on the real line
plot([scorergi, diffun(scorergi)], [-10,10])
```

```
# Scorer function Gi(z) in the complex plane
cplot(scorergi, [-8,8], [-8,8], points=50000)
```

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> scorergi(0); 1/(power(3,'7/6')*gamma('2/3'))
0.2049755424820002450503075
0.2049755424820002450503075
>>> diff(scorergi, 0); 1/(power(3,'5/6')*gamma('1/3'))
0.1494294524512754526382746
0.1494294524512754526382746
>>> scorergi(+inf); scorergi(-inf)
0.0
0.0
>>> scorergi(1)
0.2352184398104379375986902
>>> scorergi(-1)
-0.1166722172960152826494198
```

Evaluation for large arguments:

```
>>> scorergi(10)
0.03189600510067958798062034
>>> scorergi(100)
0.003183105228162961476590531
>>> scorergi(1000000)
0.00000031830988618379067217438 73
>>> 1/(pi*1000000)
```

```
0.00000031830988618379067153776 75
>>> scorergi(-1000)
-0.083582884002627803923380 14
>>> scorergi(-100000)
0.028868661186196602268095 81
>>> scorergi(50+10j)
(0.006121410279977857879098 4 - 0.0012243356764575321807479 17j)
>>> scorergi(-50-10j)
(5.2360478503522523637255 1e+29 - 3.0825422423370138148222 8e+29j)
>>> scorergi(100000j)
(-8.8066592853362310526790 25e+6474077 + 8.6847313035008355148509 62e+6474077j)
```

Verifying the connection between Gi and Hi:

```
>>> z = 0.25
>>> scorergi(z) + scorerhi(z)
0.72874690393621500786945 43
>>> airybi(z)
0.72874690393621500786945 43
```

Verifying the differential equation:

```
>>> for z in [-3.4, 0, 2.5, 1+2j]:
...     chop(diff(scorergi,z,2) - z*scorergi(z))
...
-0.31830988618379067153776 75
-0.31830988618379067153776 75
-0.31830988618379067153776 75
-0.31830988618379067153776 75
```

Verifying the integral representation:

```
>>> z = 0.5
>>> scorergi(z)
0.24472104327655819769105 39
>>> Ai,Bi = airyai,airybi
>>> Bi(z)*(Ai(inf,-1)-Ai(z,-1)) + Ai(z)*(Bi(z,-1)-Bi(0,-1))
0.24472104327655819769105 39
```

References

1. [DLMF] (page 1625) section 9.12: Scorer Functions

scorerhi()

mpmath.**scorerhi**(*z, **kwargs*)

Evaluates the second Scorer function

$$\mathrm{Hi}(z) = \mathrm{Bi}(z) \int_{-\infty}^{z} \mathrm{Ai}(t)dt - \mathrm{Ai}(z) \int_{-\infty}^{z} \mathrm{Bi}(t)dt$$

which gives a particular solution to the inhomogeneous Airy differential equation $f''(z) - zf(z) = 1/\pi$. See also `scorergi()` (page 756).

Plots

```
# Scorer function Hi(x) and Hi'(x) on the real line
plot([scorerhi, diffun(scorerhi)], [-10,2], [0,2])
```

```
# Scorer function Hi(z) in the complex plane
cplot(scorerhi, [-8,8], [-8,8], points=50000)
```

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> scorerhi(0); 2/(power(3,'7/6')*gamma('2/3'))
0.4099510849640004901006149
0.4099510849640004901006149
>>> diff(scorerhi,0); 2/(power(3,'5/6')*gamma('1/3'))
0.2988589049025509052765491
0.2988589049025509052765491
>>> scorerhi(+inf); scorerhi(-inf)
+inf
0.0
>>> scorerhi(1)
0.9722051551424333218376886
>>> scorerhi(-1)
0.2206696067929598945381098
```

Evaluation for large arguments:

```
>>> scorerhi(10)
455641153.5163291358991077
>>> scorerhi(100)
6.041223996670201399005265e+288
>>> scorerhi(1000000)
7.138269638197858094311122e+289529652
>>> scorerhi(-10)
```

```
0.031768535282502272741501 1
>>> scorerhi(-100)
0.00318309249576749986468048 3
>>> scorerhi(100j)
(-6.36619771654567212298385 7e-9 + 0.00318309886171058276168847 5j)
>>> scorerhi(50+50j)
(-5.32207626732143566929033 4e+63 + 1.47845029116524378974942 7e+65j)
>>> scorerhi(-1000-1000j)
(0.000159154943251050279656553 8 - 0.000159154943091895334973109j)
```

Verifying the differential equation:

```
>>> for z in [-3.4, 0, 2, 1+2j]:
...     chop(diff(scorerhi,z,2) - z*scorerhi(z))
...
0.318309886183790671537767 5
0.318309886183790671537767 5
0.318309886183790671537767 5
0.318309886183790671537767 5
```

Verifying the integral representation:

```
>>> z = 0.5
>>> scorerhi(z)
0.609555999826597295608994 9
>>> Ai,Bi = airyai,airybi
>>> Bi(z)*(Ai(z,-1)-Ai(-inf,-1)) - Ai(z)*(Bi(z,-1)-Bi(-inf,-1))
0.609555999826597295608994 9
```

Coulomb wave functions

coulombf()

mpmath.**coulombf**(*l*, *eta*, *z*)

Calculates the regular Coulomb wave function

$$F_l(\eta, z) = C_l(\eta) z^{l+1} e^{-iz} \, {}_1F_1(l+1-i\eta, 2l+2, 2iz)$$

where the normalization constant $C_l(\eta)$ is as calculated by `coulombc()` (page 767). This function solves the differential equation

$$f''(z) + \left(1 - \frac{2\eta}{z} - \frac{l(l+1)}{z^2}\right) f(z) = 0.$$

A second linearly independent solution is given by the irregular Coulomb wave function $G_l(\eta, z)$ (see `coulombg()` (page 765)) and thus the general solution is $f(z) = C_1 F_l(\eta, z) + C_2 G_l(\eta, z)$ for arbitrary constants C_1, C_2. Physically, the Coulomb wave functions give the radial solution to the Schrodinger equation for a point particle in a $1/z$ potential; z is then the radius and l, η are quantum numbers.

The Coulomb wave functions with real parameters are defined in Abramowitz & Stegun, section 14. However, all parameters are permitted to be complex in this implementation (see references).

Plots

```
# Regular Coulomb wave functions -- equivalent to figure 14.3 in A&S
F1 = lambda x: coulombf(0,0,x)
F2 = lambda x: coulombf(0,1,x)
F3 = lambda x: coulombf(0,5,x)
F4 = lambda x: coulombf(0,10,x)
F5 = lambda x: coulombf(0,x/2,x)
plot([F1,F2,F3,F4,F5], [0,25], [-1.2,1.6])
```

```
# Regular Coulomb wave function in the complex plane
cplot(lambda z: coulombf(1,1,z), points=50000)
```

Examples

Evaluation is supported for arbitrary magnitudes of z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> coulombf(2, 1.5, 3.5)
0.4080998961088761187426445
>>> coulombf(-2, 1.5, 3.5)
0.7103040849492536747533465
>>> coulombf(2, 1.5, '1e-10')
4.143324917492256448770769e-33
>>> coulombf(2, 1.5, 1000)
0.4482623140325567050716179
>>> coulombf(2, 1.5, 10**10)
-0.06680419643769436004661 9
```

Verifying the differential equation:

```
>>> l, eta, z = 2, 3, mpf(2.75)
>>> A, B = 1, 2
>>> f = lambda z: A*coulombf(l,eta,z) + B*coulombg(l,eta,z)
>>> chop(diff(f,z,2) + (1-2*eta/z - l*(l+1)/z**2)*f(z))
0.0
```

A Wronskian relation satisfied by the Coulomb wave functions:

```
>>> l = 2
>>> eta = 1.5
```

```
>>> F = lambda z: coulombf(l,eta,z)
>>> G = lambda z: coulombg(l,eta,z)
>>> for z in [3.5, -1, 2+3j]:
...     chop(diff(F,z)*G(z) - F(z)*diff(G,z))
...
1.0
1.0
1.0
```

Another Wronskian relation:

```
>>> F = coulombf
>>> G = coulombg
>>> for z in [3.5, -1, 2+3j]:
...     chop(F(l-1,eta,z)*G(l,eta,z)-F(l,eta,z)*G(l-1,eta,z) - l/sqrt(l**2+eta**2))
...
0.0
0.0
0.0
```

An integral identity connecting the regular and irregular wave functions:

```
>>> l, eta, z = 4+j, 2-j, 5+2j
>>> coulombf(l,eta,z) + j*coulombg(l,eta,z)
(0.799797775228403323971447 9 + 0.929448666950229551250312 7j)
>>> g = lambda t: exp(-t)*t**(l-j*eta)*(t+2*j*z)**(l+j*eta)
>>> j*exp(-j*z)*z**(-l)/fac(2*l+1)/coulombc(l,eta)*quad(g, [0,inf])
(0.799797775228403323971447 9 + 0.929448666950229551250312 7j)
```

Some test case with complex parameters, taken from Michel [2]:

```
>>> mp.dps = 15
>>> coulombf(1+0.1j, 50+50j, 100.156)
(-1.02107292320897e+15 - 2.83675545731519e+15j)
>>> coulombg(1+0.1j, 50+50j, 100.156)
(2.83675545731519e+15 - 1.02107292320897e+15j)
>>> coulombf(1e-5j, 10+1e-5j, 0.1+1e-6j)
(4.30566371247811e-14 - 9.03347835361657e-19j)
>>> coulombg(1e-5j, 10+1e-5j, 0.1+1e-6j)
(778709182061.134 + 18418936.2660553j)
```

The following reproduces a table in Abramowitz & Stegun, at twice the precision:

```
>>> mp.dps = 10
>>> eta = 2; z = 5
>>> for l in [5, 4, 3, 2, 1, 0]:
...     print("%s %s %s" % (l, coulombf(l,eta,z),
...         diff(lambda z: coulombf(l,eta,z), z)))
...
5 0.09079533488 0.1042553261
4 0.2148205331 0.2029591779
3 0.4313159311 0.320534053
2 0.7212774133 0.3952408216
1 0.9935056752 0.3708676452
0 1.143337392 0.2937960375
```

References

1. I.J. Thompson & A.R. Barnett, "Coulomb and Bessel Functions of Complex Arguments and Order", J. Comp. Phys., vol 64, no. 2, June 1986.

2.N. Michel, "Precise Coulomb wave functions for a wide range of complex l, η and z", http://arxiv.org/abs/physics/0702051v1

coulombg()

mpmath.**coulombg**(*l*, *eta*, *z*)

Calculates the irregular Coulomb wave function

$$G_l(\eta, z) = \frac{F_l(\eta, z)\cos(\chi) - F_{-l-1}(\eta, z)}{\sin(\chi)}$$

where $\chi = \sigma_l - \sigma_{-l-1} - (l + 1/2)\pi$ and $\sigma_l(\eta) = (\ln\Gamma(1 + l + i\eta) - \ln\Gamma(1 + l - i\eta))/(2i)$.

See coulombf() (page 761) for additional information.

Plots

```
# Irregular Coulomb wave functions -- equivalent to figure 14.5 in A&S
F1 = lambda x: coulombg(0,0,x)
F2 = lambda x: coulombg(0,1,x)
F3 = lambda x: coulombg(0,5,x)
F4 = lambda x: coulombg(0,10,x)
F5 = lambda x: coulombg(0,x/2,x)
plot([F1,F2,F3,F4,F5], [0,30], [-2,2])
```

```
# Irregular Coulomb wave function in the complex plane
cplot(lambda z: coulombg(1,1,z), points=50000)
```

Examples

Evaluation is supported for arbitrary magnitudes of z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> coulombg(-2, 1.5, 3.5)
1.380011900612186346255524
>>> coulombg(2, 1.5, 3.5)
1.919153700722748795245926
>>> coulombg(-2, 1.5, '1e-10')
201126715824.7329115106793
>>> coulombg(-2, 1.5, 1000)
0.18020715206911494104255 12
>>> coulombg(-2, 1.5, 10**10)
0.65210302006167807092979 4
```

The following reproduces a table in Abramowitz & Stegun, at twice the precision:

```
>>> mp.dps = 10
>>> eta = 2; z = 5
>>> for l in [1, 2, 3, 4, 5]:
...     print("%s %s %s" % (l, coulombg(l,eta,z),
...         -diff(lambda z: coulombg(l,eta,z), z)))
...
1 1.08148276 0.6028279961
2 1.496877075 0.5661803178
3 2.048694714 0.7959909551
4 3.09408669 1.731802374
```

```
5 5.629840456 4.549343289
```

Evaluation close to the singularity at $z = 0$:

```
>>> mp.dps = 15
>>> coulombg(0,10,1)
3088184933.67358
>>> coulombg(0,10,'1e-10')
5554866000719.8
>>> coulombg(0,10,'1e-100')
5554866221524.1
```

Evaluation with a half-integer value for l:

```
>>> coulombg(1.5, 1, 10)
0.852320038297334
```

coulombc()

mpmath.**coulombc**(*l*, *eta*)

Gives the normalizing Gamow constant for Coulomb wave functions,

$$C_l(\eta) = 2^l \exp\left(-\pi\eta/2 + [\ln\Gamma(1+l+i\eta) + \ln\Gamma(1+l-i\eta)]/2 - \ln\Gamma(2l+2)\right),$$

where the log gamma function with continuous imaginary part away from the negative half axis (see `loggamma()` (page 689)) is implied.

This function is used internally for the calculation of Coulomb wave functions, and automatically cached to make multiple evaluations with fixed l, η fast.

Confluent U and Whittaker functions

hyperu()

mpmath.**hyperu**(*a*, *b*, *z*)

Gives the Tricomi confluent hypergeometric function U, also known as the Kummer or confluent hypergeometric function of the second kind. This function gives a second linearly independent solution to the confluent hypergeometric differential equation (the first is provided by ${}_1F_1$ – see `hyp1f1()` (page 794)).

Examples

Evaluation for arbitrary complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyperu(2,3,4)
0.0625
>>> hyperu(0.25, 5, 1000)
0.1779949416140579573763523
>>> hyperu(0.25, 5, -1000)
(0.1256256609322773150118907 - 0.1256256609322773150118907j)
```

The U function may be singular at $z = 0$:

```
>>> hyperu(1.5, 2, 0)
+inf
>>> hyperu(1.5, -2, 0)
0.171943492128840011260367 1
```

Verifying the differential equation:

```
>>> a, b = 1.5, 2
>>> f = lambda z: hyperu(a,b,z)
>>> for z in [-10, 3, 3+4j]:
...     chop(z*diff(f,z,2) + (b-z)*diff(f,z) - a*f(z))
...
0.0
0.0
0.0
```

An integral representation:

```
>>> a,b,z = 2, 3.5, 4.25
>>> hyperu(a,b,z)
0.06674960718150520648014567
>>> quad(lambda t: exp(-z*t)*t**(a-1)*(1+t)**(b-a-1),[0,inf]) / gamma(a)
0.06674960718150520648014567
```

[1] http://people.math.sfu.ca/~cbm/aands/page_504.htm

whitm()

mpmath.**whitm**(*k*, *m*, *z*)

Evaluates the Whittaker function $M(k,m,z)$, which gives a solution to the Whittaker differential equation

$$\frac{d^2 f}{dz^2} + \left(-\frac{1}{4} + \frac{k}{z} + \frac{(\frac{1}{4} - m^2)}{z^2}\right) f = 0.$$

A second solution is given by `whitw()` (page 769).

The Whittaker functions are defined in Abramowitz & Stegun, section 13.1. They are alternate forms of the confluent hypergeometric functions ${}_1F_1$ and U:

$$M(k,m,z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} \, {}_1F_1(\tfrac{1}{2} + m - k, 1 + 2m, z)$$
$$W(k,m,z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} U(\tfrac{1}{2} + m - k, 1 + 2m, z).$$

Examples

Evaluation for arbitrary real and complex arguments is supported:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> whitm(1, 1, 1)
0.7302596799460411820509668
>>> whitm(1, 1, -1)
(0.0 - 1.417977827655098025684246j)
>>> whitm(j, j/2, 2+3j)
(3.245477713363581112736478 - 0.822879187542699127327782j)
>>> whitm(2, 3, 100000)
4.303985255686378497193063e+21707
```

Evaluation at zero:

```
>>> whitm(1,-1,0); whitm(1,-0.5,0); whitm(1,0,0)
+inf
nan
0.0
```

We can verify that whitm() (page 768) numerically satisfies the differential equation for arbitrarily chosen values:

```
>>> k = mpf(0.25)
>>> m = mpf(1.5)
>>> f = lambda z: whitm(k,m,z)
>>> for z in [-1, 2.5, 3, 1+2j]:
...     chop(diff(f,z,2) + (-0.25 + k/z + (0.25-m**2)/z**2)*f(z))
...
0.0
0.0
0.0
0.0
```

An integral involving both whitm() (page 768) and whitw() (page 769), verifying evaluation along the real axis:

```
>>> quad(lambda x: exp(-x)*whitm(3,2,x)*whitw(1,-2,x), [0,inf])
3.43886984257680022520734 1
>>> 128/(21*sqrt(pi))
3.43886984257680022520734 1
```

whitw()

mpmath.**whitw**(k, m, z)

Evaluates the Whittaker function $W(k, m, z)$, which gives a second solution to the Whittaker differential equation. (See whitm() (page 768).)

Examples

Evaluation for arbitrary real and complex arguments is supported:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> whitw(1, 1, 1)
1.195320631075811556610 12
>>> whitw(1, 1, -1)
(-0.9424875979222187313924639 - 0.2607738054097702293308689j)
>>> whitw(j, j/2, 2+3j)
(0.17828993151110338794303 69 - 0.01609578360403649340169406j)
>>> whitw(2, 3, 100000)
1.887705114889527446891274e-21705
>>> whitw(-1, -1, 100)
1.905250692824046162462058e-24
```

Evaluation at zero:

```
>>> for m in [-1, -0.5, 0, 0.5, 1]:
...     whitw(1, m, 0)
...
+inf
nan
0.0
nan
+inf
```

We can verify that whitw() (page 769) numerically satisfies the differential equation for arbitrarily chosen values:

```
>>> k = mpf(0.25)
>>> m = mpf(1.5)
>>> f = lambda z: whitw(k,m,z)
>>> for z in [-1, 2.5, 3, 1+2j]:
...     chop(diff(f,z,2) + (-0.25 + k/z + (0.25-m**2)/z**2)*f(z))
...
0.0
0.0
0.0
0.0
```

Parabolic cylinder functions

pcfd()

mpmath.**pcfd**(*n, z, **kwargs*)

Gives the parabolic cylinder function in Whittaker's notation $D_n(z) = U(-n - 1/2, z)$ (see pcfu() (page 772)). It solves the differential equation

$$y'' + \left(n + \frac{1}{2} - \frac{1}{4}z^2\right) y = 0.$$

and can be represented in terms of Hermite polynomials (see hermite() (page 783)) as

$$D_n(z) = 2^{-n/2} e^{-z^2/4} H_n\left(\frac{z}{\sqrt{2}}\right).$$

Plots

```
# Parabolic cylinder function D_n(x) on the real line for n=0,1,2,3,4
d0 = lambda x: pcfd(0,x)
d1 = lambda x: pcfd(1,x)
d2 = lambda x: pcfd(2,x)
d3 = lambda x: pcfd(3,x)
d4 = lambda x: pcfd(4,x)
plot([d0,d1,d2,d3,d4],[-7,7])
```

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> pcfd(0,0); pcfd(1,0); pcfd(2,0); pcfd(3,0)
1.0
0.0
-1.0
0.0
>>> pcfd(4,0); pcfd(-3,0)
3.0
0.6266570686577501256039413
>>> pcfd('1/2', 2+3j)
(-5.363331161232920734849056 - 3.858877821790010714163487j)
>>> pcfd(2, -10)
1.374906442631438038871515e-9
```

Verifying the differential equation:

```
>>> n = mpf(2.5)
>>> y = lambda z: pcfd(n,z)
>>> z = 1.75
>>> chop(diff(y,z,2) + (n+0.5-0.25*z**2)*y(z))
0.0
```

Rational Taylor series expansion when n is an integer:

```
>>> taylor(lambda z: pcfd(5,z), 0, 7)
[0.0, 15.0, 0.0, -13.75, 0.0, 3.96875, 0.0, -0.6015625]
```

pcfu()

mpmath.**pcfu**(*a, z, **kwargs*)

Gives the parabolic cylinder function $U(a,z)$, which may be defined for $\Re(z) > 0$ in terms of the confluent U-function (see `hyperu()` (page 767)) by

$$U(a,z) = 2^{-\frac{1}{4}-\frac{a}{2}} e^{-\frac{1}{4}z^2} U\left(\frac{a}{2}+\frac{1}{4}, \frac{1}{2}, \frac{1}{2}z^2\right)$$

or, for arbitrary z,

$$e^{-\frac{1}{4}z^2} U(a,z) = U(a,0)\,{}_1F_1\left(-\tfrac{a}{2}+\tfrac{1}{4}; \tfrac{1}{2}; -\tfrac{1}{2}z^2\right) + U'(a,0)z\,{}_1F_1\left(-\tfrac{a}{2}+\tfrac{3}{4}; \tfrac{3}{2}; -\tfrac{1}{2}z^2\right).$$

Examples

Connection to other functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> z = mpf(3)
>>> pcfu(0.5,z)
0.03210358129311151450551963
>>> sqrt(pi/2)*exp(z**2/4)*erfc(z/sqrt(2))
0.03210358129311151450551963
>>> pcfu(0.5,-z)
23.75012332835297233711255
>>> sqrt(pi/2)*exp(z**2/4)*erfc(-z/sqrt(2))
23.75012332835297233711255
>>> pcfu(0.5,-z)
23.75012332835297233711255
>>> sqrt(pi/2)*exp(z**2/4)*erfc(-z/sqrt(2))
23.75012332835297233711255
```

pcfv()

mpmath.**pcfv**(*a, z, **kwargs*)

Gives the parabolic cylinder function $V(a,z)$, which can be represented in terms of `pcfu()` (page 772) as

$$V(a,z) = \frac{\Gamma(a+\frac{1}{2})(U(a,-z) - \sin(\pi a)U(a,z)}{\pi}.$$

Examples

Wronskian relation between U and V:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a, z = 2, 3
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z)
0.7978845608028653558798921
>>> sqrt(2/pi)
0.7978845608028653558798921
>>> a, z = 2.5, 3
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z)
```

```
0.797884560802865355879892 1
>>> a, z = 0.25, -1
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z)
0.797884560802865355879892 1
>>> a, z = 2+1j, 2+3j
>>> chop(pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfv(a,z))
0.797884560802865355879892 1
```

pcfw()

mpmath.**pcfw**(*a, z, **kwargs*)

Gives the parabolic cylinder function $W(a,z)$ defined in (DLMF 12.14).

Examples

Value at the origin:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a = mpf(0.25)
>>> pcfw(a,0)
0.972283324571818076561710 4
>>> power(2,-0.75)*sqrt(abs(gamma(0.25+0.5j*a)/gamma(0.75+0.5j*a)))
0.972283324571818076561710 4
>>> diff(pcfw,(a,0),(0,1))
-0.514253394421007896600362 4
>>> -power(2,-0.25)*sqrt(abs(gamma(0.75+0.5j*a)/gamma(0.25+0.5j*a)))
-0.514253394421007896600362 4
```

Orthogonal polynomials

An orthogonal polynomial sequence is a sequence of polynomials $P_0(x), P_1(x), \ldots$ of degree $0, 1, \ldots$, which are mutually orthogonal in the sense that

$$\int_S P_n(x) P_m(x) w(x) dx = \begin{cases} c_n \neq 0 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

where S is some domain (e.g. an interval $[a,b] \in \mathbb{R}$) and $w(x)$ is a fixed *weight function*. A sequence of orthogonal polynomials is determined completely by w, S, and a normalization convention (e.g. $c_n = 1$). Applications of orthogonal polynomials include function approximation and solution of differential equations.

Orthogonal polynomials are sometimes defined using the differential equations they satisfy (as functions of x) or the recurrence relations they satisfy with respect to the order n. Other ways of defining orthogonal polynomials include differentiation formulas and generating functions. The standard orthogonal polynomials can also be represented as hypergeometric series (see *Hypergeometric functions* (page 793)), more specifically using the Gauss hypergeometric function ${}_2F_1$ in most cases. The following functions are generally implemented using hypergeometric functions since this is computationally efficient and easily generalizes.

For more information, see the Wikipedia article on orthogonal polynomials.

Legendre functions

legendre()

mpmath.**legendre**(*n, x*)

`legendre(n, x)` evaluates the Legendre polynomial $P_n(x)$. The Legendre polynomials are given by the formula

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n}(x^2-1)^n.$$

Alternatively, they can be computed recursively using

$$P_0(x) = 1$$
$$P_1(x) = x$$
$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x).$$

A third definition is in terms of the hypergeometric function ${}_2F_1$, whereby they can be generalized to arbitrary n:

$$P_n(x) = {}_2F_1\left(-n, n+1, 1, \frac{1-x}{2}\right)$$

Plots

```
# Legendre polynomials P_n(x) on [-1,1] for n=0,1,2,3,4
f0 = lambda x: legendre(0,x)
f1 = lambda x: legendre(1,x)
f2 = lambda x: legendre(2,x)
f3 = lambda x: legendre(3,x)
f4 = lambda x: legendre(4,x)
plot([f0,f1,f2,f3,f4],[-1,1])
```

Basic evaluation

The Legendre polynomials assume fixed values at the points $x = -1$ and $x = 1$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint([legendre(n, 1) for n in range(6)])
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> nprint([legendre(n, -1) for n in range(6)])
[1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
```

The coefficients of Legendre polynomials can be recovered using degree-n Taylor expansion:

```
>>> for n in range(5):
...     nprint(chop(taylor(lambda x: legendre(n, x), 0, n)))
...
[1.0]
[0.0, 1.0]
[-0.5, 0.0, 1.5]
[0.0, -1.5, 0.0, 2.5]
[0.375, 0.0, -3.75, 0.0, 4.375]
```

The roots of Legendre polynomials are located symmetrically on the interval $[-1, 1]$:

```
>>> for n in range(5):
...     nprint(polyroots(taylor(lambda x: legendre(n, x), 0, n)[::-1]))
...
[]
[0.0]
[-0.57735, 0.57735]
[-0.774597, 0.0, 0.774597]
[-0.861136, -0.339981, 0.339981, 0.861136]
```

An example of an evaluation for arbitrary n:

```
>>> legendre(0.75, 2+4j)
(1.94952805264875 + 2.1071073099422j)
```

Orthogonality

The Legendre polynomials are orthogonal on $[-1, 1]$ with respect to the trivial weight $w(x) = 1$. That is, $P_m(x)P_n(x)$ integrates to zero if $m \neq n$ and to $2/(2n + 1)$ if $m = n$:

```
>>> m, n = 3, 4
>>> quad(lambda x: legendre(m,x)*legendre(n,x), [-1, 1])
0.0
>>> m, n = 4, 4
>>> quad(lambda x: legendre(m,x)*legendre(n,x), [-1, 1])
0.222222222222222
```

Differential equation

The Legendre polynomials satisfy the differential equation

$$((1 - x^2)y')' + n(n + 1)y' = 0.$$

We can verify this numerically:

```
>>> n = 3.6
>>> x = 0.73
>>> P = legendre
```

```
>>> A = diff(lambda t: (1-t**2)*diff(lambda u: P(n,u), t), x)
>>> B = n*(n+1)*P(n,x)
>>> nprint(A+B,1)
9.0e-16
```

legenp()

mpmath.**legenp**(*n, m, z, type=2*)

Calculates the (associated) Legendre function of the first kind of degree *n* and order *m*, $P_n^m(z)$. Taking $m = 0$ gives the ordinary Legendre function of the first kind, $P_n(z)$. The parameters may be complex numbers.

In terms of the Gauss hypergeometric function, the (associated) Legendre function is defined as

$$P_n^m(z) = \frac{1}{\Gamma(1-m)} \frac{(1+z)^{m/2}}{(1-z)^{m/2}} \, {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right).$$

With *type=3* instead of *type=2*, the alternative definition

$$\hat{P}_n^m(z) = \frac{1}{\Gamma(1-m)} \frac{(z+1)^{m/2}}{(z-1)^{m/2}} \, {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right).$$

is used. These functions correspond respectively to `LegendreP[n,m,2,z]` and `LegendreP[n,m,3,z]` in Mathematica.

The general solution of the (associated) Legendre differential equation

$$(1-z^2)f''(z) - 2zf'(z) + \left(n(n+1) - \frac{m^2}{1-z^2}\right) f(z) = 0$$

is given by $C_1 P_n^m(z) + C_2 Q_n^m(z)$ for arbitrary constants C_1, C_2, where $Q_n^m(z)$ is a Legendre function of the second kind as implemented by `legenq()` (page 777).

Examples

Evaluation for arbitrary parameters and arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> legenp(2, 0, 10); legendre(2, 10)
149.5
149.5
>>> legenp(-2, 0.5, 2.5)
(1.972260393822275434196053 - 1.972260393822275434196053j)
>>> legenp(2+3j, 1-j, -0.5+4j)
(-3.335677248386698208736542 - 5.663270217461022307645625j)
>>> chop(legenp(3, 2, -1.5, type=2))
28.125
>>> chop(legenp(3, 2, -1.5, type=3))
-28.125
```

Verifying the associated Legendre differential equation:

```
>>> n, m = 2, -0.5
>>> C1, C2 = 1, -3
>>> f = lambda z: C1*legenp(n,m,z) + C2*legenq(n,m,z)
>>> deq = lambda z: (1-z**2)*diff(f,z,2) - 2*z*diff(f,z) + \
...     (n*(n+1)-m**2/(1-z**2))*f(z)
>>> for z in [0, 2, -1.5, 0.5+2j]:
```

```
...     chop(deq(mpmathify(z)))
...
0.0
0.0
0.0
0.0
```

legenq()

mpmath.**legenq**(*n, m, z, type=2*)

Calculates the (associated) Legendre function of the second kind of degree *n* and order *m*, $Q_n^m(z)$. Taking $m = 0$ gives the ordinary Legendre function of the second kind, $Q_n(z)$. The parameters may complex numbers.

The Legendre functions of the second kind give a second set of solutions to the (associated) Legendre differential equation. (See legenp() (page 776).) Unlike the Legendre functions of the first kind, they are not polynomials of z for integer n, m but rational or logarithmic functions with poles at $z = \pm 1$.

There are various ways to define Legendre functions of the second kind, giving rise to different complex structure. A version can be selected using the *type* keyword argument. The *type=2* and *type=3* functions are given respectively by

$$Q_n^m(z) = \frac{\pi}{2\sin(\pi m)}\left(\cos(\pi m)P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)}P_n^{-m}(z)\right)$$
$$\hat{Q}_n^m(z) = \frac{\pi}{2\sin(\pi m)}e^{\pi i m}\left(\hat{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)}\hat{P}_n^{-m}(z)\right)$$

where P and $\hat{P}$ are the *type=2* and *type=3* Legendre functions of the first kind. The formulas above should be understood as limits when m is an integer.

These functions correspond to `LegendreQ[n,m,2,z]` (or `LegendreQ[n,m,z]`) and `LegendreQ[n,m,3,z]` in Mathematica. The *type=3* function is essentially the same as the function defined in Abramowitz & Stegun (eq. 8.1.3) but with $(z+1)^{m/2}(z-1)^{m/2}$ instead of $(z^2-1)^{m/2}$, giving slightly different branches.

Examples

Evaluation for arbitrary parameters and arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> legenq(2, 0, 0.5)
-0.8186632680417568557122028
>>> legenq(-1.5, -2, 2.5)
(0.6655964618250228714288277 + 0.3937692045497259717762649j)
>>> legenq(2-j, 3+4j, -6+5j)
(-10001.95256487468541686564 - 6011.691337610097577791134j)
```

Different versions of the function:

```
>>> legenq(2, 1, 0.5)
0.7298060598018049369381857
>>> legenq(2, 1, 1.5)
(-7.902916572420817192300921 + 0.19986500726059766007245 02j)
>>> legenq(2, 1, 0.5, type=3)
(2.040524284763495081918338 - 0.7298060598018049369381857j)
>>> chop(legenq(2, 1, 1.5, type=3))
-0.19986500726059766007245 02
```

Chebyshev polynomials

chebyt()

mpmath.**chebyt**(*n*, *x*)

`chebyt(n, x)` evaluates the Chebyshev polynomial of the first kind $T_n(x)$, defined by the identity

$$T_n(\cos x) = \cos(nx).$$

The Chebyshev polynomials of the first kind are a special case of the Jacobi polynomials, and by extension of the hypergeometric function ${}_2F_1$. They can thus also be evaluated for nonintegral n.

Plots

```
# Chebyshev polynomials T_n(x) on [-1,1] for n=0,1,2,3,4
f0 = lambda x: chebyt(0,x)
f1 = lambda x: chebyt(1,x)
f2 = lambda x: chebyt(2,x)
f3 = lambda x: chebyt(3,x)
f4 = lambda x: chebyt(4,x)
plot([f0,f1,f2,f3,f4],[-1,1])
```

Basic evaluation

The coefficients of the n-th polynomial can be recovered using using degree-n Taylor expansion:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(chop(taylor(lambda x: chebyt(n, x), 0, n)))
...
[1.0]
[0.0, 1.0]
[-1.0, 0.0, 2.0]
[0.0, -3.0, 0.0, 4.0]
[1.0, 0.0, -8.0, 0.0, 8.0]
```

Orthogonality

The Chebyshev polynomials of the first kind are orthogonal on the interval $[-1,1]$ with respect to the weight function $w(x) = 1/\sqrt{1-x^2}$:

```
>>> f = lambda x: chebyt(m,x)*chebyt(n,x)/sqrt(1-x**2)
>>> m, n = 3, 4
>>> nprint(quad(f, [-1, 1]),1)
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.57079632596448
```

chebyu()

mpmath.**chebyu**(*n*, *x*)

`chebyu(n, x)` evaluates the Chebyshev polynomial of the second kind $U_n(x)$, defined by the identity

$$U_n(\cos x) = \frac{\sin((n+1)x)}{\sin(x)}.$$

The Chebyshev polynomials of the second kind are a special case of the Jacobi polynomials, and by extension of the hypergeometric function ${}_2F_1$. They can thus also be evaluated for nonintegral n.

Plots

```
# Chebyshev polynomials U_n(x) on [-1,1] for n=0,1,2,3,4
f0 = lambda x: chebyu(0,x)
f1 = lambda x: chebyu(1,x)
f2 = lambda x: chebyu(2,x)
f3 = lambda x: chebyu(3,x)
f4 = lambda x: chebyu(4,x)
plot([f0,f1,f2,f3,f4],[-1,1])
```

Basic evaluation

The coefficients of the n-th polynomial can be recovered using using degree-n Taylor expansion:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(chop(taylor(lambda x: chebyu(n, x), 0, n)))
...
[1.0]
[0.0, 2.0]
[-1.0, 0.0, 4.0]
[0.0, -4.0, 0.0, 8.0]
[1.0, 0.0, -12.0, 0.0, 16.0]
```

Orthogonality

The Chebyshev polynomials of the second kind are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = \sqrt{1 - x^2}$:

```
>>> f = lambda x: chebyu(m,x)*chebyu(n,x)*sqrt(1-x**2)
>>> m, n = 3, 4
>>> quad(f, [-1, 1])
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.5707963267949
```

Jacobi polynomials

jacobi()

mpmath.**jacobi**(*n*, *a*, *b*, *z*)

`jacobi(n, a, b, x)` evaluates the Jacobi polynomial $P_n^{(a,b)}(x)$. The Jacobi polynomials are a special case of the hypergeometric function ${}_2F_1$ given by:

$$P_n^{(a,b)}(x) = \binom{n+a}{n} {}_2F_1\left(-n, 1+a+b+n, a+1, \frac{1-x}{2}\right).$$

Note that this definition generalizes to nonintegral values of n. When n is an integer, the hypergeometric series terminates after a finite number of terms, giving a polynomial in x.

Evaluation of Jacobi polynomials

A special evaluation is $P_n^{(a,b)}(1) = \binom{n+a}{n}$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> jacobi(4, 0.5, 0.25, 1)
2.4609375
>>> binomial(4+0.5, 4)
2.4609375
```

A Jacobi polynomial of degree n is equal to its Taylor polynomial of degree n. The explicit coefficients of Jacobi polynomials can therefore be recovered easily using `taylor()` (page 911):

```
>>> for n in range(5):
...     nprint(taylor(lambda x: jacobi(n,1,2,x), 0, n))
...
[1.0]
[-0.5, 2.5]
[-0.75, -1.5, 5.25]
[0.5, -3.5, -3.5, 10.5]
[0.625, 2.5, -11.25, -7.5, 20.625]
```

For nonintegral n, the Jacobi "polynomial" is no longer a polynomial:

```
>>> nprint(taylor(lambda x: jacobi(0.5,1,2,x), 0, 4))
[0.309983, 1.84119, -1.26933, 1.26699, -1.34808]
```

Orthogonality

The Jacobi polynomials are orthogonal on the interval $[-1,1]$ with respect to the weight function $w(x) = (1-x)^a(1+x)^b$. That is, $w(x)P_n^{(a,b)}(x)P_m^{(a,b)}(x)$ integrates to zero if $m \neq n$ and to a nonzero number if $m = n$.

The orthogonality is easy to verify using numerical quadrature:

```
>>> P = jacobi
>>> f = lambda x: (1-x)**a * (1+x)**b * P(m,a,b,x) * P(n,a,b,x)
>>> a = 2
>>> b = 3
>>> m, n = 3, 4
>>> chop(quad(f, [-1, 1]), 1)
0.0
>>> m, n = 4, 4
```

```
>>> quad(f, [-1, 1])
1.9047619047619
```

Differential equation

The Jacobi polynomials are solutions of the differential equation

$$(1-x^2)y'' + (b-a-(a+b+2)x)y' + n(n+a+b+1)y = 0.$$

We can verify that `jacobi()` (page 781) approximately satisfies this equation:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> a = 2.5
>>> b = 4
>>> n = 3
>>> y = lambda x: jacobi(n,a,b,x)
>>> x = pi
>>> A0 = n*(n+a+b+1)*y(x)
>>> A1 = (b-a-(a+b+2)*x)*diff(y,x)
>>> A2 = (1-x**2)*diff(y,x,2)
>>> nprint(A2 + A1 + A0, 1)
4.0e-12
```

The difference of order 10^{-12} is as close to zero as it could be at 15-digit working precision, since the terms are large:

```
>>> A0, A1, A2
(26560.2328981879, -21503.7641037294, -5056.46879445852)
```

Gegenbauer polynomials

gegenbauer()

mpmath.**gegenbauer**(*n*, *a*, *z*)

Evaluates the Gegenbauer polynomial, or ultraspherical polynomial,

$$C_n^{(a)}(z) = \binom{n+2a-1}{n} {}_2F_1\left(-n, n+2a; a+\frac{1}{2}; \frac{1}{2}(1-z)\right).$$

When n is a nonnegative integer, this formula gives a polynomial in z of degree n, but all parameters are permitted to be complex numbers. With $a = 1/2$, the Gegenbauer polynomial reduces to a Legendre polynomial.

Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> gegenbauer(3, 0.5, -10)
-2485.0
>>> gegenbauer(1000, 10, 100)
3.012757178975667428359374e+2322
>>> gegenbauer(2+3j, -0.75, -1000j)
(-5038991.358609026523401901 + 9414549.285447104177860806j)
```

Evaluation at negative integer orders:

```
>>> gegenbauer(-4, 2, 1.75)
-1.0
>>> gegenbauer(-4, 3, 1.75)
0.0
>>> gegenbauer(-4, 2j, 1.75)
0.0
>>> gegenbauer(-7, 0.5, 3)
8989.0
```

The Gegenbauer polynomials solve the differential equation:

```
>>> n, a = 4.5, 1+2j
>>> f = lambda z: gegenbauer(n, a, z)
>>> for z in [0, 0.75, -0.5j]:
...     chop((1-z**2)*diff(f,z,2) - (2*a+1)*z*diff(f,z) + n*(n+2*a)*f(z))
...
0.0
0.0
0.0
```

The Gegenbauer polynomials have generating function $(1-2zt+t^2)^{-a}$:

```
>>> a, z = 2.5, 1
>>> taylor(lambda t: (1-2*z*t+t**2)**(-a), 0, 3)
[1.0, 5.0, 15.0, 35.0]
>>> [gegenbauer(n,a,z) for n in range(4)]
[1.0, 5.0, 15.0, 35.0]
```

The Gegenbauer polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1-z^2)^{a-\frac{1}{2}}$:

```
>>> a, n, m = 2.5, 4, 5
>>> Cn = lambda z: gegenbauer(n, a, z, zeroprec=1000)
>>> Cm = lambda z: gegenbauer(m, a, z, zeroprec=1000)
>>> chop(quad(lambda z: Cn(z)*Cm(z)*(1-z**2)*(a-0.5), [-1, 1]))
0.0
```

Hermite polynomials

hermite()

mpmath.**hermite**(n, z)

Evaluates the Hermite polynomial $H_n(z)$, which may be defined using the recurrence

$$H_0(z) = 1$$
$$H_1(z) = 2z$$
$$H_{n+1} = 2zH_n(z) - 2nH_{n-1}(z).$$

The Hermite polynomials are orthogonal on $(-\infty,\infty)$ with respect to the weight e^{-z^2}. More generally, allowing arbitrary complex values of n, the Hermite function $H_n(z)$ is defined as

$$H_n(z) = (2z)^n \, {}_2F_0\left(-\frac{n}{2}, \frac{1-n}{2}, -\frac{1}{z^2}\right)$$

for $\Re z > 0$, or generally

$$H_n(z) = 2^n \sqrt{\pi} \left(\frac{1}{\Gamma\left(\frac{1-n}{2}\right)} {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma\left(-\frac{n}{2}\right)} {}_1F_1\left(\frac{1-n}{2}, \frac{3}{2}, z^2\right) \right).$$

Plots

```
# Hermite polynomials H_n(x) on the real line for n=0,1,2,3,4
f0 = lambda x: hermite(0,x)
f1 = lambda x: hermite(1,x)
f2 = lambda x: hermite(2,x)
f3 = lambda x: hermite(3,x)
f4 = lambda x: hermite(4,x)
plot([f0,f1,f2,f3,f4],[-2,2],[-25,25])
```

Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hermite(0, 10)
1.0
>>> hermite(1, 10); hermite(2, 10)
20.0
398.0
>>> hermite(10000, 2)
4.950440066552087387515653e+19334
>>> hermite(3, -10**8)
```

```
-79999999999999998800000000.0
>>> hermite(-3, -10**8)
1.67515975172987768292030 1e+4342944819032534
>>> hermite(2+3j, -1+2j)
(-0.076521306029935133894219 01 - 0.108466244996191458027600 7j)
```

Coefficients of the first few Hermite polynomials are:

```
>>> for n in range(7):
...     chop(taylor(lambda z: hermite(n, z), 0, n))
...
[1.0]
[0.0, 2.0]
[-2.0, 0.0, 4.0]
[0.0, -12.0, 0.0, 8.0]
[12.0, 0.0, -48.0, 0.0, 16.0]
[0.0, 120.0, 0.0, -160.0, 0.0, 32.0]
[-120.0, 0.0, 720.0, 0.0, -480.0, 0.0, 64.0]
```

Values at $z = 0$:

```
>>> for n in range(-5, 9):
...     hermite(n, 0)
...
0.027694591420398687926533 87
0.083333333333333333333333333
0.221556731363189503412270 9
0.5
0.886226925452758013649083 7
1.0
0.0
-2.0
0.0
12.0
0.0
-120.0
0.0
1680.0
```

Hermite functions satisfy the differential equation:

```
>>> n = 4
>>> f = lambda z: hermite(n, z)
>>> z = 1.5
>>> chop(diff(f,z,2) - 2*z*diff(f,z) + 2*n*f(z))
0.0
```

Verifying orthogonality:

```
>>> chop(quad(lambda t: hermite(2,t)*hermite(4,t)*exp(-t**2), [-inf,inf]))
0.0
```

Laguerre polynomials

laguerre()

mpmath.**laguerre**(*n*, *a*, *z*)

Gives the generalized (associated) Laguerre polynomial, defined by

$$L_n^a(z) = \frac{\Gamma(n+b+1)}{\Gamma(b+1)\Gamma(n+1)}\,{}_1F_1(-n, a+1, z).$$

With $a = 0$ and n a nonnegative integer, this reduces to an ordinary Laguerre polynomial, the sequence of which begins $L_0(z) = 1, L_1(z) = 1 - z, L_2(z) = z^2 - 2z + 1, \ldots$.

The Laguerre polynomials are orthogonal with respect to the weight $z^a e^{-z}$ on $[0, \infty)$.

Plots

```
# Hermite polynomials L_n(x) on the real line for n=0,1,2,3,4
f0 = lambda x: laguerre(0,0,x)
f1 = lambda x: laguerre(1,0,x)
f2 = lambda x: laguerre(2,0,x)
f3 = lambda x: laguerre(3,0,x)
f4 = lambda x: laguerre(4,0,x)
plot([f0,f1,f2,f3,f4],[0,10],[-10,10])
```

Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> laguerre(5, 0, 0.25)
0.03726399739583333333333333
>>> laguerre(1+j, 0.5, 2+3j)
```

```
(4.474921610704496808379097 - 11.02058050372068958069241j)
>>> laguerre(2, 0, 10000)
49980001.0
>>> laguerre(2.5, 0, 10000)
-9.327764910194842158583189e+4328
```

The first few Laguerre polynomials, normalized to have integer coefficients:

```
>>> for n in range(7):
...     chop(taylor(lambda z: fac(n)*laguerre(n, 0, z), 0, n))
...
[1.0]
[1.0, -1.0]
[2.0, -4.0, 1.0]
[6.0, -18.0, 9.0, -1.0]
[24.0, -96.0, 72.0, -16.0, 1.0]
[120.0, -600.0, 600.0, -200.0, 25.0, -1.0]
[720.0, -4320.0, 5400.0, -2400.0, 450.0, -36.0, 1.0]
```

Verifying orthogonality:

```
>>> Lm = lambda t: laguerre(m,a,t)
>>> Ln = lambda t: laguerre(n,a,t)
>>> a, n, m = 2.5, 2, 3
>>> chop(quad(lambda t: exp(-t)*t**a*Lm(t)*Ln(t), [0,inf]))
0.0
```

Spherical harmonics

spherharm()

mpmath.**spherharm**(*l, m, theta, phi*)

Evaluates the spherical harmonic $Y_l^m(\theta,\phi)$,

$$Y_l^m(\theta,\phi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta) e^{im\phi}$$

where P_l^m is an associated Legendre function (see `legenp()` (page 776)).

Here $\theta \in [0,\pi]$ denotes the polar coordinate (ranging from the north pole to the south pole) and $\phi \in [0,2\pi]$ denotes the azimuthal coordinate on a sphere. Care should be used since many different conventions for spherical coordinate variables are used.

Usually spherical harmonics are considered for $l \in \mathbb{N}$, $m \in \mathbb{Z}$, $|m| \le l$. More generally, l, m, θ, ϕ are permitted to be complex numbers.

Note: `spherharm()` (page 787) returns a complex number, even the value is purely real.

Plots

```
# Real part of spherical harmonic Y_(4,0)(theta,phi)
def Y(l,m):
    def g(theta,phi):
        R = abs(fp.re(fp.spherharm(l,m,theta,phi)))
        x = R*fp.cos(phi)*fp.sin(theta)
        y = R*fp.sin(phi)*fp.sin(theta)
        z = R*fp.cos(theta)
```

```
        return [x,y,z]
    return g

fp.splot(Y(4,0), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,0), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,1), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,2), [0,fp.pi], [0,2*fp.pi], points=300)
# fp.splot(Y(4,3), [0,fp.pi], [0,2*fp.pi], points=300)
```

$Y_{4,0}$:

$Y_{4,1}$:

$Y_{4,2}$:

$Y_{4,3}$:

$Y_{4,4}$:

Examples

Some low-order spherical harmonics with reference values:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> theta = pi/4
>>> phi = pi/3
>>> spherharm(0,0,theta,phi); 0.5*sqrt(1/pi)*expj(0)
(0.2820947917738781434740397 + 0.0j)
(0.2820947917738781434740397 + 0.0j)
>>> spherharm(1,-1,theta,phi); 0.5*sqrt(3/(2*pi))*expj(-phi)*sin(theta)
(0.1221506279757299803965962 - 0.2115710938304086076055298j)
(0.1221506279757299803965962 - 0.2115710938304086076055298j)
>>> spherharm(1,0,theta,phi); 0.5*sqrt(3/pi)*cos(theta)*expj(0)
(0.3454941494713354792652446 + 0.0j)
(0.3454941494713354792652446 + 0.0j)
>>> spherharm(1,1,theta,phi); -0.5*sqrt(3/(2*pi))*expj(phi)*sin(theta)
(-0.1221506279757299803965962 - 0.2115710938304086076055298j)
(-0.1221506279757299803965962 - 0.2115710938304086076055298j)
```

With the normalization convention used, the spherical harmonics are orthonormal on the unit sphere:

```
>>> sphere = [0,pi], [0,2*pi]
>>> dS = lambda t,p: fp.sin(t)   # differential element
>>> Y1 = lambda t,p: fp.spherharm(l1,m1,t,p)
>>> Y2 = lambda t,p: fp.conj(fp.spherharm(l2,m2,t,p))
```

```
>>> l1 = l2 = 3; m1 = m2 = 2
>>> print(fp.quad(lambda t,p: Y1(t,p)*Y2(t,p)*dS(t,p), *sphere))
(1+0j)
>>> m2 = 1    # m1 != m2
>>> print(fp.chop(fp.quad(lambda t,p: Y1(t,p)*Y2(t,p)*dS(t,p), *sphere)))
0.0
```

Evaluation is accurate for large orders:

```
>>> spherharm(1000,750,0.5,0.25)
(3.776445785304252879026585e-102 - 5.824412787718347944934 84e-102j)
```

Evaluation works with complex parameter values:

```
>>> spherharm(1+j, 2j, 2+3j, -0.5j)
(64.4492233111375999215499 2 + 1981.69391984140808968174 3j)
```

Hypergeometric functions

The functions listed in *Exponential integrals and error functions* (page 700), *Bessel functions and related functions* (page 715) and *Orthogonal polynomials* (page 773), and many other functions as well, are merely particular instances of the generalized hypergeometric function ${}_pF_q$. The functions listed in the following section enable efficient direct evaluation of the underlying hypergeometric series, as well as linear combinations, limits with respect to parameters, and analytic continuations thereof. Extensions to twodimensional series are also provided. See also the basic or q-analog of the hypergeometric series in *q-functions* (page 871).

For convenience, most of the hypergeometric series of low order are provided as standalone functions. They can equivalently be evaluated using `hyper()` (page 800). As will be demonstrated in the respective docstrings, all the `hyp#f#` functions implement analytic continuations and/or asymptotic expansions with respect to the argument z, thereby permitting evaluation for z anywhere in the complex plane. Functions of higher degree can be computed via `hyper()` (page 800), but generally only in rapidly convergent instances.

Most hypergeometric and hypergeometric-derived functions accept optional keyword arguments to specify options for `hypercomb()` or `hyper()`. Some useful options are *maxprec*, *maxterms*, *zeroprec*, *accurate_small*, *hmag*, *force_series*, *asymp_tol* and *eliminate*. These options give control over what to do in case of slow convergence, extreme loss of accuracy or evaluation at zeros (these two cases cannot generally be distinguished from each other automatically), and singular parameter combinations.

Common hypergeometric series

hyp0f1()

mpmath.**hyp0f1**(a, z)

Gives the hypergeometric function ${}_0F_1$, sometimes known as the confluent limit function, defined as

$$ {}_0F_1(a,z) = \sum_{k=0}^{\infty} \frac{1}{(a)_k} \frac{z^k}{k!}. $$

This function satisfies the differential equation $zf''(z) + af'(z) = f(z)$, and is related to the Bessel function of the first kind (see `besselj()` (page 715)).

`hyp0f1(a,z)` is equivalent to `hyper([],[a],z)`; see documentation for `hyper()` (page 800) for more information.

Examples

Evaluation for arbitrary arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp0f1(2, 0.25)
1.130318207984970054415392
>>> hyp0f1((1,2), 1234567)
6.272871875462270560462 7e+964
>>> hyp0f1(3+4j, 1000000j)
(3.905169561300910030267132e+606 + 3.807708544441684513934213e+606j)
```

Evaluation is supported for arbitrarily large values of z, using asymptotic expansions:

```
>>> hyp0f1(1, 10**50)
2.131705322874965310390701e+8685889638065036553022565
>>> hyp0f1(1, -10**50)
1.115945364792025420300208e-13
```

Verifying the differential equation:

```
>>> a = 2.5
>>> f = lambda z: hyp0f1(a,z)
>>> for z in [0, 10, 3+4j]:
...     chop(z*diff(f,z,2) + a*diff(f,z) - f(z))
...
0.0
0.0
0.0
```

hyp1f1()

mpmath.**hyp1f1**(*a, b, z*)

Gives the confluent hypergeometric function of the first kind,

$$ {}_1F_1(a,b,z) = \sum_{k=0}^{\infty} \frac{(a)_k}{(b)_k} \frac{z^k}{k!}, $$

also known as Kummer's function and sometimes denoted by $M(a,b,z)$. This function gives one solution to the confluent (Kummer's) differential equation

$$ zf''(z) + (b-z)f'(z) - af(z) = 0. $$

A second solution is given by the U function; see `hyperu()` (page 767). Solutions are also given in an alternate form by the Whittaker functions (`whitm()` (page 768), `whitw()` (page 769)).

`hyp1f1(a,b,z)` is equivalent to `hyper([a],[b],z)`; see documentation for `hyper()` (page 800) for more information.

Examples

Evaluation for real and complex values of the argument z, with fixed parameters $a = 2, b = -1/3$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp1f1(2, (-1,3), 3.25)
-2815.956856924817275640248
>>> hyp1f1(2, (-1,3), -3.25)
-1.145036502407444445553107
>>> hyp1f1(2, (-1,3), 1000)
-8.021799872770764149793693e+441
>>> hyp1f1(2, (-1,3), -1000)
0.0000031319876330068135945353 31
>>> hyp1f1(2, (-1,3), 100+100j)
(-3.189190365227034385898282e+48 - 1.106169926814270418999315e+49j)
```

Parameters may be complex:

```
>>> hyp1f1(2+3j, -1+j, 10j)
(261.897790518104514267335 1 + 160.8930312845682213562172j)
```

Arbitrarily large values of z are supported:

```
>>> hyp1f1(3, 4, 10**20)
3.890569218254486878220752e+43429448190325182745
>>> hyp1f1(3, 4, -10**20)
6.0e-60
>>> hyp1f1(3, 4, 10**20*j)
(-1.935753855797342532571597e-20 - 2.291911213325184901239155e-20j)
```

Verifying the differential equation:

```
>>> a, b = 1.5, 2
>>> f = lambda z: hyp1f1(a,b,z)
>>> for z in [0, -10, 3, 3+4j]:
...     chop(z*diff(f,z,2) + (b-z)*diff(f,z) - a*f(z))
...
0.0
0.0
0.0
0.0
```

An integral representation:

```
>>> a, b = 1.5, 3
>>> z = 1.5
>>> hyp1f1(a,b,z)
2.269381460919952778587441
>>> g = lambda t: exp(z*t)*t**(a-1)*(1-t)**(b-a-1)
>>> gammaprod([b],[a,b-a])*quad(g, [0,1])
2.269381460919952778587441
```

hyp1f2()

mpmath.**hyp1f2**(*a1, b1, b2, z*)

Gives the hypergeometric function ${}_1F_2(a_1,a_2;b_1,b_2;z)$. The call `hyp1f2(a1,b1,b2,z)` is equivalent to `hyper([a1],[b1,b2],z)`.

Evaluation works for complex and arbitrarily large arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
```

```
>>> a, b, c = 1.5, (-1,3), 2.25
>>> hyp1f2(a, b, c, 10**20)
-1.159388148811981535941434e+8685889639
>>> hyp1f2(a, b, c, -10**20)
-12.60262607892655945795907
>>> hyp1f2(a, b, c, 10**20*j)
(4.237220401382240876065501e+6141851464 - 2.950930337531768015892987e+6141851464j)
>>> hyp1f2(2+3j, -2j, 0.5j, 10-20j)
(135881.9905586966432662004 - 86681.95885418079535738828j)
```

hyp2f0()

mpmath.**hyp2f0**(*a*, *b*, *z*)

Gives the hypergeometric function ${}_2F_0$, defined formally by the series

$$ {}_2F_0(a,b;;z) = \sum_{n=0}^{\infty} (a)_n (b)_n \frac{z^n}{n!}. $$

This series usually does not converge. For small enough z, it can be viewed as an asymptotic series that may be summed directly with an appropriate truncation. When this is not the case, `hyp2f0()` (page 796) gives a regularized sum, or equivalently, it uses a representation in terms of the hypergeometric U function [1]. The series also converges when either a or b is a nonpositive integer, as it then terminates into a polynomial after $-a$ or $-b$ terms.

Examples

Evaluation is supported for arbitrary complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp2f0((2,3), 1.25, -100)
0.07095851870980052763312791
>>> hyp2f0((2,3), 1.25, 100)
(-0.03254379032170590665041131 + 0.07269254613282301012735797j)
>>> hyp2f0(-0.75, 1-j, 4j)
(-0.357998703108273226486215 5 - 3.052951783922142735255881j)
```

Even with real arguments, the regularized value of 2F0 is often complex-valued, but the imaginary part decreases exponentially as $z \to 0$. In the following example, the first call uses complex evaluation while the second has a small enough z to evaluate using the direct series and thus the returned value is strictly real (this should be taken to indicate that the imaginary part is less than `eps`):

```
>>> mp.dps = 15
>>> hyp2f0(1.5, 0.5, 0.05)
(1.04166637647907 + 8.34584913683906e-8j)
>>> hyp2f0(1.5, 0.5, 0.0005)
1.00037535207621
```

The imaginary part can be retrieved by increasing the working precision:

```
>>> mp.dps = 80
>>> nprint(hyp2f0(1.5, 0.5, 0.009).imag)
1.23828e-46
```

In the polynomial case (the series terminating), 2F0 can evaluate exactly:

```
>>> mp.dps = 15
>>> hyp2f0(-6,-6,2)
291793.0
>>> identify(hyp2f0(-2,1,0.25))
'(5/8)'
```

The coefficients of the polynomials can be recovered using Taylor expansion:

```
>>> nprint(taylor(lambda x: hyp2f0(-3,0.5,x), 0, 10))
[1.0, -1.5, 2.25, -1.875, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint(taylor(lambda x: hyp2f0(-4,0.5,x), 0, 10))
[1.0, -2.0, 4.5, -7.5, 6.5625, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

[1] http://people.math.sfu.ca/~cbm/aands/page_504.htm

hyp2f1()

mpmath.**hyp2f1**(*a, b, c, z*)

Gives the Gauss hypergeometric function ${}_2F_1$ (often simply referred to as *the* hypergeometric function), defined for $|z| < 1$ as

$$ {}_2F_1(a,b,c,z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!}. $$

and for $|z| \geq 1$ by analytic continuation, with a branch cut on $(1, \infty)$ when necessary.

Special cases of this function include many of the orthogonal polynomials as well as the incomplete beta function and other functions. Properties of the Gauss hypergeometric function are documented comprehensively in many references, for example Abramowitz & Stegun, section 15.

The implementation supports the analytic continuation as well as evaluation close to the unit circle where $|z| \approx 1$. The syntax `hyp2f1(a,b,c,z)` is equivalent to `hyper([a,b],[c],z)`.

Examples

Evaluation with z inside, outside and on the unit circle, for fixed parameters:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp2f1(2, (1,2), 4, 0.75)
1.303703703703703703703704
>>> hyp2f1(2, (1,2), 4, -1.75)
0.7431290566046919177853916
>>> hyp2f1(2, (1,2), 4, 1.75)
(1.418075801749271137026239 - 1.114976146679907015775102j)
>>> hyp2f1(2, (1,2), 4, 1)
1.6
>>> hyp2f1(2, (1,2), 4, -1)
0.8235498012182875315037882
>>> hyp2f1(2, (1,2), 4, j)
(0.9144026291433065674259078 + 0.2050415770437884900574923j)
>>> hyp2f1(2, (1,2), 4, 2+j)
(0.9274013540258103029011549 + 0.7455257875808100868984496j)
>>> hyp2f1(2, (1,2), 4, 0.25j)
(0.9931169055799728251931672 + 0.06154836525312066938147793j)
```

Evaluation with complex parameter values:

```
>>> hyp2f1(1+j, 0.75, 10j, 1+5j)
(0.883483331971347992338963 8 + 0.705388688064810506834350 9j)
```

Evaluation with $z = 1$:

```
>>> hyp2f1(-2.5, 3.5, 1.5, 1)
0.0
>>> hyp2f1(-2.5, 3, 4, 1)
0.06926406926406926406926407
>>> hyp2f1(2, 3, 4, 1)
+inf
```

Evaluation for huge arguments:

```
>>> hyp2f1((-1,3), 1.75, 4, '1e100')
(7.883714220959876246415651e+32 + 1.365499358305579597618785e+33j)
>>> hyp2f1((-1,3), 1.75, 4, '1e1000000')
(7.883714220959876246415651e+333332 + 1.365499358305579597618785e+333333j)
>>> hyp2f1((-1,3), 1.75, 4, '1e1000000j')
(1.365499358305579597618785e+333333 - 7.883714220959876246415651e+333332j)
```

An integral representation:

```
>>> a,b,c,z = -0.5, 1, 2.5, 0.25
>>> g = lambda t: t**(b-1) * (1-t)**(c-b-1) * (1-t*z)**(-a)
>>> gammaprod([c],[b,c-b]) * quad(g, [0,1])
0.948045881436282447885261 8
>>> hyp2f1(a,b,c,z)
0.948045881436282447885261 8
```

Verifying the hypergeometric differential equation:

```
>>> f = lambda z: hyp2f1(a,b,c,z)
>>> chop(z*(1-z)*diff(f,z,2) + (c-(a+b+1)*z)*diff(f,z) - a*b*f(z))
0.0
```

hyp2f2()

mpmath.**hyp2f2**(*a1, a2, b1, b2, z*)

Gives the hypergeometric function ${}_2F_2(a_1, a_2; b_1, b_2; z)$. The call `hyp2f2(a1,a2,b1,b2,z)` is equivalent to `hyper([a1,a2],[b1,b2],z)`.

Evaluation works for complex and arbitrarily large arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a, b, c, d = 1.5, (-1,3), 2.25, 4
>>> hyp2f2(a, b, c, d, 10**20)
-5.275758229007902299823821e+43429448190325182663
>>> hyp2f2(a, b, c, d, -10**20)
2561445.079983207701073448
>>> hyp2f2(a, b, c, d, 10**20*j)
(2218276.509664121194836667 - 1280722.539991603850462856j)
>>> hyp2f2(2+3j, -2j, 0.5j, 4j, 10-20j)
(80500.68321405666957342788 - 20346.82752982813540993502j)
```

hyp2f3()

mpmath.**hyp2f3**(*a1, a2, b1, b2, b3, z*)

Gives the hypergeometric function ${}_2F_3(a_1, a_2; b_1, b_2, b_3; z)$. The call `hyp2f3(a1,a2,b1,b2,b3,z)` is equivalent to `hyper([a1,a2],[b1,b2,b3],z)`.

Evaluation works for arbitrarily large arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a1,a2,b1,b2,b3 = 1.5, (-1,3), 2.25, 4, (1,5)
>>> hyp2f3(a1,a2,b1,b2,b3,10**20)
-4.169178177065714963568963e+8685889590
>>> hyp2f3(a1,a2,b1,b2,b3,-10**20)
7064472.587757755088178629
>>> hyp2f3(a1,a2,b1,b2,b3,10**20*j)
(-5.163368465314934589818543e+6141851415 + 1.783578125755972803440364e+6141851416j)
>>> hyp2f3(2+3j, -2j, 0.5j, 4j, -1-j, 10-20j)
(-2280.938956687033150740228 + 13620.97336609573659199632j)
>>> hyp2f3(2+3j, -2j, 0.5j, 4j, -1-j, 10000000-20000000j)
(4.849835186175096516193e+3504 - 3.365981529122220091353633e+3504j)
```

hyp3f2()

mpmath.**hyp3f2**(*a1, a2, a3, b1, b2, z*)

Gives the generalized hypergeometric function ${}_3F_2$, defined for $|z| < 1$ as

$$ {}_3F_2(a_1, a_2, a_3, b_1, b_2, z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k (a_3)_k}{(b_1)_k (b_2)_k} \frac{z^k}{k!}. $$

and for $|z| \geq 1$ by analytic continuation. The analytic structure of this function is similar to that of ${}_2F_1$, generally with a singularity at $z = 1$ and a branch cut on $(1, \infty)$.

Evaluation is supported inside, on, and outside the circle of convergence $|z| = 1$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp3f2(1,2,3,4,5,0.25)
1.083533123380934241548707
>>> hyp3f2(1,2+2j,3,4,5,-10+10j)
(0.157465106600600463291436 1 - 0.031942090218852264008929 63j)
>>> hyp3f2(1,2,3,4,5,-10)
0.3071141169208772603266489
>>> hyp3f2(1,2,3,4,5,10)
(-0.4857045320523947050581423 - 0.5988311440454888436888028j)
>>> hyp3f2(0.25,1,1,2,1.5,1)
1.157370995096772047567631
>>> (8-pi-2*ln2)/3
1.157370995096772047567631
>>> hyp3f2(1+j,0.5j,2,1,-2j,-1)
(1.745184906150294864759 59 + 0.1454701525056682297614029j)
>>> hyp3f2(1+j,0.5j,2,1,-2j,sqrt(j))
(0.9829816481834277511138055 - 0.4059040020276937085081127j)
>>> hyp3f2(-3,2,1,-5,4,1)
1.41
>>> hyp3f2(-3,2,1,-5,4,2)
2.12
```

Evaluation very close to the unit circle:

```
>>> hyp3f2(1,2,3,4,5,'1.0001')
(1.564877796743282766872279 - 3.76821518787438186031973e-11j)
>>> hyp3f2(1,2,3,4,5,'1+0.0001j')
(1.564747153061671573212831 + 0.00013057575703660845576484 82j)
>>> hyp3f2(1,2,3,4,5,'0.9999')
1.564616644881686134983664
>>> hyp3f2(1,2,3,4,5,'-0.9999')
0.782389625346167806019 6207
```

Note: Evaluation for $|z-1|$ small can currently be inaccurate or slow for some parameter combinations.

For various parameter combinations, ${}_3F_2$ admits representation in terms of hypergeometric functions of lower degree, or in terms of simpler functions:

```
>>> for a, b, z in [(1,2,-1), (2,0.5,1)]:
...     hyp2f1(a,b,a+b+0.5,z)**2
...     hyp3f2(2*a,a+b,2*b,a+b+0.5,2*a+2*b,z)
...
0.42461044619664390060863 08
0.42461044619664390060863 08
7.11111111111111111111111
7.11111111111111111111111

>>> z = 2+3j
>>> hyp3f2(0.5,1,1.5,2,2,z)
(0.762144093924334241972914 4 + 0.424911773505803764991572 3j)
>>> 4*(pi-2*ellipe(z))/(pi*z)
(0.762144093924334241972914 4 + 0.424911773505803764991572 3j)
```

Generalized hypergeometric functions

hyper()

mpmath.**hyper**(*a_s*, *b_s*, *z*)

Evaluates the generalized hypergeometric function

$$ {}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n \ldots (a_p)_n}{(b_1)_n (b_2)_n \ldots (b_q)_n} \frac{z^n}{n!} $$

where $(x)_n$ denotes the rising factorial (see `rf()` (page 690)).

The parameters lists `a_s` and `b_s` may contain integers, real numbers, complex numbers, as well as exact fractions given in the form of tuples (p,q). `hyper()` (page 800) is optimized to handle integers and fractions more efficiently than arbitrary floating-point parameters (since rational parameters are by far the most common).

Examples

Verifying that `hyper()` (page 800) gives the sum in the definition, by comparison with `nsum()` (page 882):

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a,b,c,d = 2,3,4,5
>>> x = 0.25
```

```
>>> hyper([a,b],[c,d],x)
1.07890394116493487608623 7
>>> fn = lambda n: rf(a,n)*rf(b,n)/rf(c,n)/rf(d,n)*x**n/fac(n)
>>> nsum(fn, [0, inf])
1.07890394116493487608623 7
```

The parameters can be any combination of integers, fractions, floats and complex numbers:

```
>>> a, b, c, d, e = 1, (-1,2), pi, 3+4j, (2,3)
>>> x = 0.2j
>>> hyper([a,b],[c,d,e],x)
(0.992357161643402481083188 7 - 0.00575384873388387974299312 2j)
>>> b, e = -0.5, mpf(2)/3
>>> fn = lambda n: rf(a,n)*rf(b,n)/rf(c,n)/rf(d,n)/rf(e,n)*x**n/fac(n)
>>> nsum(fn, [0, inf])
(0.992357161643402481083188 7 - 0.00575384873388387974299312 2j)
```

The ${}_0F_0$ and ${}_1F_0$ series are just elementary functions:

```
>>> a, z = sqrt(2), +pi
>>> hyper([],[],z)
23.1406926327792690057290 9
>>> exp(z)
23.1406926327792690057290 9
>>> hyper([a],[],z)
(-0.0906913287992292016033411 4 + 0.328322432394616208357965 6j)
>>> (1-z)**(-a)
(-0.0906913287992292016033411 4 + 0.328322432394616208357965 6j)
```

If any a_k coefficient is a nonpositive integer, the series terminates into a finite polynomial:

```
>>> hyper([1,1,1,-3],[2,5],1)
0.790476190476190476190476 2
>>> identify(_)
'(83/105)'
```

If any b_k is a nonpositive integer, the function is undefined (unless the series terminates before the division by zero occurs):

```
>>> hyper([1,1,1,-3],[-2,5],1)
Traceback (most recent call last):
  ...
ZeroDivisionError: pole in hypergeometric series
>>> hyper([1,1,1,-1],[-2,5],1)
1.1
```

Except for polynomial cases, the radius of convergence R of the hypergeometric series is either $R = \infty$ (if $p \leq q$), $R = 1$ (if $p = q + 1$), or $R = 0$ (if $p > q + 1$).

The analytic continuations of the functions with $p = q + 1$, i.e. ${}_2F_1$, ${}_3F_2$, ${}_4F_3$, etc, are all implemented and therefore these functions can be evaluated for $|z| \geq 1$. The shortcuts `hyp2f1()` (page 797), `hyp3f2()` (page 799) are available to handle the most common cases (see their documentation), but functions of higher degree are also supported via `hyper()` (page 800):

```
>>> hyper([1,2,3,4], [5,6,7], 1)   # 4F3 at finite-valued branch point
1.14178350552687073131142 3
>>> hyper([4,5,6,7], [1,2,3], 1)   # 4F3 at pole
+inf
```

```
>>> hyper([1,2,3,4,5], [6,7,8,9], 10)    # 5F4
(1.54399891652797225971725 7 - 0.587630992958040802881636 5j)
>>> hyper([1,2,3,4,5,6], [7,8,9,10,11], 1j)   # 6F5
(0.999656582185357906350246 6 + 0.012972107590563060444566 9j)
```

Near $z = 1$ with noninteger parameters:

```
>>> hyper(['1/3',1,'3/2',2], ['1/5','11/6','41/8'], 1)
2.2194333522355861212500 27
>>> hyper(['1/3',1,'3/2',2], ['1/5','11/6','5/4'], 1)
+inf
>>> eps1 = extradps(6)(lambda: 1 - mpf('1e-6'))()
>>> hyper(['1/3',1,'3/2',2], ['1/5','11/6','5/4'], eps1)
2923978034.412973409330956
```

Please note that, as currently implemented, evaluation of ${}_pF_{p-1}$ with $p \geq 3$ may be slow or inaccurate when $|z - 1|$ is small, for some parameter values.

When $p > q+1$, `hyper` computes the (iterated) Borel sum of the divergent series. For ${}_2F_0$ the Borel sum has an analytic solution and can be computed efficiently (see `hyp2f0()` (page 796)). For higher degrees, the functions is evaluated first by attempting to sum it directly as an asymptotic series (this only works for tiny $|z|$), and then by evaluating the Borel regularized sum using numerical integration. Except for special parameter combinations, this can be extremely slow.

```
>>> hyper([1,1], [], 0.5)          # regularization of 2F0
(1.3409654195801465620864 48 + 0.8503366631752726568782447j)
>>> hyper([1,1,1,1], [1], 0.5)     # regularization of 4F1
(1.1082872136894751458306 99 + 0.5327107430640678181200491j)
```

With the following magnitude of argument, the asymptotic series for ${}_3F_1$ gives only a few digits. Using Borel summation, `hyper` can produce a value with full accuracy:

```
>>> mp.dps = 15
>>> hyper([2,0.5,4], [5.25], '0.08', force_series=True)
Traceback (most recent call last):
  ...
NoConvergence: Hypergeometric series converges too slowly. Try increasing maxterms.
>>> hyper([2,0.5,4], [5.25], '0.08', asymp_tol=1e-4)
1.0725535790737
>>> hyper([2,0.5,4], [5.25], '0.08')
(1.07269542893559 + 5.54668863216891e-5j)
>>> hyper([2,0.5,4], [5.25], '-0.08', asymp_tol=1e-4)
0.946344925484879
>>> hyper([2,0.5,4], [5.25], '-0.08')
0.946312503737771
>>> mp.dps = 25
>>> hyper([2,0.5,4], [5.25], '-0.08')
0.9463125037377662296700858
```

Note that with the positive z value, there is a complex part in the correct result, which falls below the tolerance of the asymptotic series.

hypercomb()

mpmath.**hypercomb**(*ctx*, *function*, *params*=[ ], *discard_known_zeros*=*True*, ***kwargs*)

Computes a weighted combination of hypergeometric functions

$$\sum_{r=1}^{N} \left[\prod_{k=1}^{l_r} {w_{r,k}}^{c_{r,k}} \frac{\prod_{k=1}^{m_r} \Gamma(\alpha_{r,k})}{\prod_{k=1}^{n_r} \Gamma(\beta_{r,k})} \, {}_{p_r}F_{q_r}(a_{r,1}, \ldots, a_{r,p}; b_{r,1}, \ldots, b_{r,q}; z_r) \right].$$

Typically the parameters are linear combinations of a small set of base parameters; `hypercomb()` (page 802) permits computing a correct value in the case that some of the α, β, b turn out to be nonpositive integers, or if division by zero occurs for some w^c, assuming that there are opposing singularities that cancel out. The limit is computed by evaluating the function with the base parameters perturbed, at a higher working precision.

The first argument should be a function that takes the perturbable base parameters `params` as input and returns N tuples `(w, c, alpha, beta, a, b, z)`, where the coefficients `w`, `c`, gamma factors `alpha`, `beta`, and hypergeometric coefficients `a`, `b` each should be lists of numbers, and `z` should be a single number.

Examples

The following evaluates

$$(a-1)\frac{\Gamma(a-3)}{\Gamma(a-4)} \, {}_1F_1(a, a-1, z) = e^z(a-4)(a+z-1)$$

with $a=1, z=3$. There is a zero factor, two gamma function poles, and the 1F1 function is singular; all singularities cancel out to give a finite value:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> hypercomb(lambda a: [([a-1],[1],[a-3],[a-4],[a],[a-1],3)], [1])
-180.769832308689
>>> -9*exp(3)
-180.769832308689
```

Meijer G-function

meijerg()

mpmath.**meijerg**(*a_s*, *b_s*, *z*, *r=1*, ***kwargs*)

Evaluates the Meijer G-function, defined as

$$G_{p,q}^{m,n}\left(\begin{matrix} a_1, \ldots, a_n; a_{n+1} \ldots a_p \\ b_1, \ldots, b_m; b_{m+1} \ldots b_q \end{matrix} \middle| \; z; r\right) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^{m} \Gamma(b_j+s) \prod_{j=1}^{n} \Gamma(1-a_j-s)}{\prod_{j=n+1}^{p} \Gamma(a_j+s) \prod_{j=m+1}^{q} \Gamma(1-b_j-s)} z^{-s/r} ds$$

for an appropriate choice of the contour L (see references).

There are p elements a_j. The argument *a_s* should be a pair of lists, the first containing the n elements $a_1, \ldots, a_n$ and the second containing the $p-n$ elements $a_{n+1}, \ldots a_p$.

There are q elements b_j. The argument *b_s* should be a pair of lists, the first containing the m elements $b_1, \ldots, b_m$ and the second containing the $q-m$ elements $b_{m+1}, \ldots b_q$.

The implicit tuple (m, n, p, q) constitutes the order or degree of the Meijer G-function, and is determined by the lengths of the coefficient vectors. Confusingly, the indices in this tuple appear in a different order from the coefficients, but this notation is standard. The many examples given below should hopefully clear up any potential confusion.

Algorithm

The Meijer G-function is evaluated as a combination of hypergeometric series. There are two versions of the function, which can be selected with the optional *series* argument.

series=1 uses a sum of m ${}_pF_{q-1}$ functions of z

series=2 uses a sum of n ${}_qF_{p-1}$ functions of $1/z$

The default series is chosen based on the degree and $|z|$ in order to be consistent with Mathematica's. This definition of the Meijer G-function has a discontinuity at $|z| = 1$ for some orders, which can be avoided by explicitly specifying a series.

Keyword arguments are forwarded to `hypercomb()` (page 802).

Examples

Many standard functions are special cases of the Meijer G-function (possibly rescaled and/or with branch cut corrections). We define some test parameters:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a = mpf(0.75)
>>> b = mpf(1.5)
>>> z = mpf(2.25)
```

The exponential function: $e^z = G^{1,0}_{0,1}\left(\begin{matrix} - \\ 0 \end{matrix}\middle| -z\right)$

```
>>> meijerg([[],[]], [[0],[]], -z)
9.48773583635852572055036 9
>>> exp(z)
9.48773583635852572055036 9
```

The natural logarithm: $\log(1+z) = G^{1,2}_{2,2}\left(\begin{matrix} 1, 1 \\ 1, 0 \end{matrix}\middle| -z\right)$

```
>>> meijerg([[1,1],[]], [[1],[0]], z)
1.17865499634164611721902 3
>>> log(1+z)
1.17865499634164611721902 3
```

A rational function: $\frac{z}{z+1} = G^{1,2}_{2,2}\left(\begin{matrix} 1, 1 \\ 1, 1 \end{matrix}\middle| z\right)$

```
>>> meijerg([[1,1],[]], [[1],[1]], z)
0.692307692307692307692307 7
>>> z/(z+1)
0.692307692307692307692307 7
```

The sine and cosine functions:

$\frac{1}{\sqrt{\pi}} \sin(2\sqrt{z}) = G^{1,0}_{0,2}\left(\begin{matrix} - \\ \frac{1}{2}, 0 \end{matrix}\middle| z\right)$

$\frac{1}{\sqrt{\pi}} \cos(2\sqrt{z}) = G^{1,0}_{0,2}\left(\begin{matrix} - \\ 0, \frac{1}{2} \end{matrix}\middle| z\right)$

```
>>> meijerg([[],[]], [[0.5],[0]], (z/2)**2)
0.438980792921867668229645 3
>>> sin(z)/sqrt(pi)
0.438980792921867668229645 3
>>> meijerg([[],[]], [[0],[0.5]], (z/2)**2)
-0.354409014599627542333176 2
>>> cos(z)/sqrt(pi)
-0.354409014599627542333176 2
```

Bessel functions:

$$J_a(2\sqrt{z}) = G_{0,2}^{1,0}\left(\begin{matrix} - \\ \frac{a}{2}, -\frac{a}{2} \end{matrix}\middle| z\right)$$

$$Y_a(2\sqrt{z}) = G_{1,3}^{2,0}\left(\begin{matrix} \frac{-a-1}{2} \\ \frac{a}{2}, -\frac{a}{2}, \frac{-a-1}{2} \end{matrix}\middle| z\right)$$

$$(-z)^{a/2} z^{-a/2} I_a(2\sqrt{z}) = G_{0,2}^{1,0}\left(\begin{matrix} - \\ \frac{a}{2}, -\frac{a}{2} \end{matrix}\middle| -z\right)$$

$$2K_a(2\sqrt{z}) = G_{0,2}^{2,0}\left(\begin{matrix} - \\ \frac{a}{2}, -\frac{a}{2} \end{matrix}\middle| z\right)$$

As the example with the Bessel I function shows, a branch factor is required for some arguments when inverting the square root.

```
>>> meijerg([[],[]], [[a/2],[-a/2]], (z/2)**2)
0.50594257895971548585272 64
>>> besselj(a,z)
0.50594257895971548585272 64
>>> meijerg([[],[(-a-1)/2]], [[a/2,-a/2],[(-a-1)/2]], (z/2)**2)
0.18538689500665569414425 59
>>> bessely(a, z)
0.18538689500665569414425 59
>>> meijerg([[],[]], [[a/2],[-a/2]], -(z/2)**2)
(0.86859133224276538757174 76 + 2.0969649744601992005517 38j)
>>> (-z)**(a/2) / z**(a/2) * besseli(a, z)
(0.86859133224276538757174 76 + 2.0969649744601992005517 38j)
>>> 0.5*meijerg([[],[]], [[a/2,-a/2],[]], (z/2)**2)
0.09334163695597828403796071
>>> besselk(a,z)
0.09334163695597828403796071
```

Error functions:

$$\sqrt{\pi} z^{2(a-1)} \mathrm{erfc}(z) = G_{1,2}^{2,0}\left(\begin{matrix} a \\ a-1, a-\frac{1}{2} \end{matrix}\middle| z, \frac{1}{2}\right)$$

```
>>> meijerg([[],[a]], [[a-1,a-0.5],[]], z, 0.5)
0.0017283984312309195746871 2
>>> sqrt(pi) * z**(2*a-2) * erfc(z)
0.0017283984312309195746871 2
```

A Meijer G-function of higher degree, (1,1,2,3):

```
>>> meijerg([[a],[b]], [[a],[b,a-1]], z)
1.5598446744305021011561 7
>>> sin((b-a)*pi)/pi*(exp(z)-1)*z**(a-1)
1.5598446744305021011561 7
```

A Meijer G-function of still higher degree, (4,1,2,4), that can be expanded as a messy combination of exponential integrals:

```
>>> meijerg([[a],[2*b-a]], [[b,a,b-0.5,-1-a+2*b],[]], z)
0.33236671336585572718980 61
>>> chop(4**(a-b+1)*sqrt(pi)*gamma(2*b-2*a)*z**a*\
...     expint(2*b-2*a, -2*sqrt(-z))*expint(2*b-2*a, 2*sqrt(-z)))
0.33236671336585572718980 61
```

In the following case, different series give different values:

```
>>> chop(meijerg([[1],[0.25]],[[3],[0.5]],-2))
-0.06417628097442437076207337
```

```
>>> meijerg([[1],[0.25]],[[3],[0.5]],-2,series=1)
0.14286994261551175118730 47
>>> chop(meijerg([[1],[0.25]],[[3],[0.5]],-2,series=2))
-0.0641762809744243707620 7337
```

References

1. http://en.wikipedia.org/wiki/Meijer_G-function
2. http://mathworld.wolfram.com/MeijerG-Function.html
3. http://functions.wolfram.com/HypergeometricFunctions/MeijerG/
4. http://functions.wolfram.com/HypergeometricFunctions/MeijerG1/

Bilateral hypergeometric series

bihyper()

mpmath.**bihyper**(*a_s, b_s, z, **kwargs*)

Evaluates the bilateral hypergeometric series

$$ {}_AH_B(a_1,\dots,a_k;b_1,\dots,b_B;z) = \sum_{n=-\infty}^{\infty} \frac{(a_1)_n \dots (a_A)_n}{(b_1)_n \dots (b_B)_n} z^n $$

where, for direct convergence, $A = B$ and $|z| = 1$, although a regularized sum exists more generally by considering the bilateral series as a sum of two ordinary hypergeometric functions. In order for the series to make sense, none of the parameters may be integers.

Examples

The value of ${}_2H_2$ at $z = 1$ is given by Dougall's formula:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> a,b,c,d = 0.5, 1.5, 2.25, 3.25
>>> bihyper([a,b],[c,d],1)
-14.4911802621234578614884 7
>>> gammaprod([c,d,1-a,1-b,c+d-a-b-1],[c-a,d-a,c-b,d-b])
-14.4911802621234578614884 7
```

The regularized function ${}_1H_0$ can be expressed as the sum of one ${}_2F_0$ function and one ${}_1F_1$ function:

```
>>> a = mpf(0.25)
>>> z = mpf(0.75)
>>> bihyper([a], [], z)
(0.24543933896572738413855 82 + 0.24543933896572738413855 82j)
>>> hyper([a,1],[],z) + (hyper([1],[1-a],-1/z)-1)
(0.24543933896572738413855 82 + 0.24543933896572738413855 82j)
>>> hyper([a,1],[],z) + hyper([1],[2-a],-1/z)/z/(a-1)
(0.24543933896572738413855 82 + 0.24543933896572738413855 82j)
```

References

1. [Slater] (page 1625) (chapter 6: "Bilateral Series", pp. 180-189)
2. [Wikipedia] (page 1626) http://en.wikipedia.org/wiki/Bilateral_hypergeometric_series

Hypergeometric functions of two variables

hyper2d()

mpmath.**hyper2d**(*a, b, x, y, **kwargs*)

Sums the generalized 2D hypergeometric series

$$\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{P((a),m,n)}{Q((b),m,n)}\frac{x^m y^n}{m!n!}$$

where $(a)=(a_1,\ldots,a_r)$, $(b)=(b_1,\ldots,b_s)$ and where P and Q are products of rising factorials such as $(a_j)_n$ or $(a_j)_{m+n}$. P and Q are specified in the form of dicts, with the m and n dependence as keys and parameter lists as values. The supported rising factorials are given in the following table (note that only a few are supported in Q):

Key	Rising factorial	Q
'm'	$(a_j)_m$	Yes
'n'	$(a_j)_n$	Yes
'm+n'	$(a_j)_{m+n}$	Yes
'm-n'	$(a_j)_{m-n}$	No
'n-m'	$(a_j)_{n-m}$	No
'2m+n'	$(a_j)_{2m+n}$	No
'2m-n'	$(a_j)_{2m-n}$	No
'2n-m'	$(a_j)_{2n-m}$	No

For example, the Appell F1 and F4 functions

$$F_1=\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{(a)_{m+n}(b)_m(c)_n}{(d)_{m+n}}\frac{x^m y^n}{m!n!}$$

$$F_4=\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{(a)_{m+n}(b)_{m+n}}{(c)_m(d)_n}\frac{x^m y^n}{m!n!}$$

can be represented respectively as

```
hyper2d({'m+n':[a], 'm':[b], 'n':[c]}, {'m+n':[d]}, x, y)
```

```
hyper2d({'m+n':[a,b]}, {'m':[c], 'n':[d]}, x, y)
```

More generally, hyper2d() (page 807) can evaluate any of the 34 distinct convergent second-order (generalized Gaussian) hypergeometric series enumerated by Horn, as well as the Kampe de Feriet function.

The series is computed by rewriting it so that the inner series (i.e. the series containing n and y) has the form of an ordinary generalized hypergeometric series and thereby can be evaluated efficiently using hyper() (page 800). If possible, manually swapping x and y and the corresponding parameters can sometimes give better results.

Examples

Two separable cases: a product of two geometric series, and a product of two Gaussian hypergeometric functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> x, y = mpf(0.25), mpf(0.5)
>>> hyper2d({'m':1,'n':1}, {}, x,y)
2.6666666666666666666667
>>> 1/(1-x)/(1-y)
2.6666666666666666666667
>>> hyper2d({'m':[1,2],'n':[3,4]}, {'m':[5],'n':[6]}, x,y)
4.1643585312389383196698 56
>>> hyp2f1(1,2,5,x)*hyp2f1(3,4,6,y)
4.1643585312389383196698 56
```

Some more series that can be done in closed form:

```
>>> hyper2d({'m':1,'n':1},{'m+n':1},x,y)
2.013417124712514809623881
>>> (exp(x)*x-exp(y)*y)/(x-y)
2.013417124712514809623881
```

Six of the 34 Horn functions, G1-G3 and H1-H3:

```
>>> from mpmath import *
>>> mp.dps = 10; mp.pretty = True
>>> x, y = 0.0625, 0.125
>>> a1,a2,b1,b2,c1,c2,d = 1.1,-1.2,-1.3,-1.4,1.5,-1.6,1.7
>>> hyper2d({'m+n':a1,'n-m':b1,'m-n':b2},{},x,y)  # G1
1.139090746
>>> nsum(lambda m,n: rf(a1,m+n)*rf(b1,n-m)*rf(b2,m-n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
1.139090746
>>> hyper2d({'m':a1,'n':a2,'n-m':b1,'m-n':b2},{},x,y)  # G2
0.9503682696
>>> nsum(lambda m,n: rf(a1,m)*rf(a2,n)*rf(b1,n-m)*rf(b2,m-n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
0.9503682696
>>> hyper2d({'2n-m':a1,'2m-n':a2},{},x,y)  # G3
1.029372029
>>> nsum(lambda m,n: rf(a1,2*n-m)*rf(a2,2*m-n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
1.029372029
>>> hyper2d({'m-n':a1,'m+n':b1,'n':c1},{'m':d},x,y)  # H1
-1.605331256
>>> nsum(lambda m,n: rf(a1,m-n)*rf(b1,m+n)*rf(c1,n)/rf(d,m)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
-1.605331256
>>> hyper2d({'m-n':a1,'m':b1,'n':[c1,c2]},{'m':d},x,y)  # H2
-2.35405404
>>> nsum(lambda m,n: rf(a1,m-n)*rf(b1,m)*rf(c1,n)*rf(c2,n)/rf(d,m)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
-2.35405404
>>> hyper2d({'2m+n':a1,'n':b1},{'m+n':c1},x,y)  # H3
0.974479074
>>> nsum(lambda m,n: rf(a1,2*m+n)*rf(b1,n)/rf(c1,m+n)*\
...     x**m*y**n/fac(m)/fac(n), [0,inf], [0,inf])
0.974479074
```

References

1. [SrivastavaKarlsson] (page 1625)
2. [Weisstein] (page 1626) http://mathworld.wolfram.com/HornFunction.html
3. [Weisstein] (page 1626) http://mathworld.wolfram.com/AppellHypergeometricFunction.html

appellf1()

mpmath.**appellf1**(*a, b1, b2, c, x, y, **kwargs*)

Gives the Appell F1 hypergeometric function of two variables,

$$F_1(a,b_1,b_2,c,x,y) = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a)_{m+n}(b_1)_m(b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m!n!}.$$

This series is only generally convergent when $|x| < 1$ and $|y| < 1$, although appellf1() (page 808) can evaluate an analytic continuation with respecto to either variable, and sometimes both.

Examples

Evaluation is supported for real and complex parameters:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf1(1,0,0.5,1,0.5,0.25)
1.154700538379251529018298
>>> appellf1(1,1+j,0.5,1,0.5,0.5j)
(1.138403860350148085179415 + 1.510544741058517621110615j)
```

For some integer parameters, the F1 series reduces to a polynomial:

```
>>> appellf1(2,-4,-3,1,2,5)
-816.0
>>> appellf1(-5,1,2,1,4,5)
-20528.0
```

The analytic continuation with respect to either x or y, and sometimes with respect to both, can be evaluated:

```
>>> appellf1(2,3,4,5,100,0.5)
(0.00062310427141653292797386 62 + 0.000000576914927714842577449985 7j)
>>> appellf1('1.1', '0.3', '0.2+2j', '0.4', '0.2', 1.5+3j)
(-0.178260456689395489712870 2 + 0.00247240710454621611716149 9j)
>>> appellf1(1,2,3,4,10,12)
-0.071229938300667763749293 13
```

For certain arguments, F1 reduces to an ordinary hypergeometric function:

```
>>> appellf1(1,2,3,5,0.5,0.25)
1.547902270302684019335555
>>> 4*hyp2f1(1,2,5,'1/3')/3
1.547902270302684019335555
>>> appellf1(1,2,3,4,0,1.5)
(-1.717202506168937502740238 - 2.792526803190927323077905j)
>>> hyp2f1(1,3,4,1.5)
(-1.717202506168937502740238 - 2.792526803190927323077905j)
```

The F1 function satisfies a system of partial differential equations:

```
>>> a,b1,b2,c,x,y = map(mpf, [1,0.5,0.25,1.125,0.25,-0.25])
>>> F = lambda x,y: appellf1(a,b1,b2,c,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) +
...      y*(1-x)*diff(F,(x,y),(1,1)) +
...      (c-(a+b1+1)*x)*diff(F,(x,y),(1,0)) -
...      b1*y*diff(F,(x,y),(0,1)) -
...      a*b1*F(x,y))
0.0
>>>
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) +
...      x*(1-y)*diff(F,(x,y),(1,1)) +
...      (c-(a+b2+1)*y)*diff(F,(x,y),(0,1)) -
...      b2*x*diff(F,(x,y),(1,0)) -
...      a*b2*F(x,y))
0.0
```

The Appell F1 function allows for closed-form evaluation of various integrals, such as any integral of the form $\int x^r(x+a)^p(x+b)^q dx$:

```
>>> def integral(a,b,p,q,r,x1,x2):
...     a,b,p,q,r,x1,x2 = map(mpmathify, [a,b,p,q,r,x1,x2])
...     f = lambda x: x**r * (x+a)**p * (x+b)**q
...     def F(x):
...         v = x**(r+1)/(r+1) * (a+x)**p * (b+x)**q
...         v *= (1+x/a)**(-p)
...         v *= (1+x/b)**(-q)
...         v *= appellf1(r+1,-p,-q,2+r,-x/a,-x/b)
...         return v
...     print("Num. quad: %s" % quad(f, [x1,x2]))
...     print("Appell F1: %s" % (F(x2)-F(x1)))
...
>>> integral('1/5','4/3','-2','3','1/2',0,1)
Num. quad: 9.07333535878577620657698 1
Appell F1: 9.07333535878577620657698 1
>>> integral('3/2','4/3','-2','3','1/2',0,1)
Num. quad: 1.09282917199962645434467 8
Appell F1: 1.09282917199962645434467 8
>>> integral('3/2','4/3','-2','3','1/2',12,25)
Num. quad: 1106.32322504023511649892 7
Appell F1: 1106.32322504023511649892 7
```

Also incomplete elliptic integrals fall into this category [1]:

```
>>> def E(z, m):
...     if (pi/2).ae(z):
...         return ellipe(m)
...     return 2*round(re(z)/pi)*ellipe(m) + mpf(-1)**round(re(z)/pi)*\
...         sin(z)*appellf1(0.5,0.5,-0.5,1.5,sin(z)**2,m*sin(z)**2)
...
>>> z, m = 1, 0.5
>>> E(z,m); quad(lambda t: sqrt(1-m*sin(t)**2), [0,pi/4,3*pi/4,z])
0.927329883624440066965904 2
0.927329883624440066965904 2
>>> z, m = 3, 2
>>> E(z,m); quad(lambda t: sqrt(1-m*sin(t)**2), [0,pi/4,3*pi/4,z])
(1.05749575233723422971583 6 + 1.19814023473559220743992 2j)
(1.05749575233723422971583 6 + 1.19814023473559220743992 2j)
```

References

1. [WolframFunctions] (page 1626) http://functions.wolfram.com/EllipticIntegrals/EllipticE2/26/01/
2. [SrivastavaKarlsson] (page 1625)
3. [CabralRosetti] (page 1625)
4. [Vidunas] (page 1626)
5. [Slater] (page 1625)

appellf2()

mpmath.**appellf2**(*a, b1, b2, c1, c2, x, y, **kwargs*)

Gives the Appell F2 hypergeometric function of two variables

$$F_2(a,b_1,b_2,c_1,c_2,x,y) = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a)_{m+n}(b_1)_m(b_2)_n}{(c_1)_m(c_2)_n} \frac{x^m y^n}{m!n!}.$$

The series is generally absolutely convergent for $|x| + |y| < 1$.

Examples

Evaluation for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf2(1,2,3,4,5,0.25,0.125)
1.257417193533135344785602
>>> appellf2(1,-3,-4,2,3,2,3)
-42.8
>>> appellf2(0.5,0.25,-0.25,2,3,0.25j,0.25)
(0.9880539519421899867041719 + 0.01497616165031102661476978j)
>>> chop(appellf2(1,1+j,1-j,3j,-3j,0.25,0.25))
1.201311219287411337955192
>>> appellf2(1,1,1,4,6,0.125,16)
(-0.09455532250274744282125152 - 0.7647282253046207836769297j)
```

A transformation formula:

```
>>> a,b1,b2,c1,c2,x,y = map(mpf, [1,2,0.5,0.25,1.625,-0.125,0.125])
>>> appellf2(a,b1,b2,c1,c2,x,y)
0.2299211717841180783309688
>>> (1-x)**(-a)*appellf2(a,c1-b1,b2,c1,c2,x/(x-1),y/(1-x))
0.2299211717841180783309688
```

A system of partial differential equations satisfied by F2:

```
>>> a,b1,b2,c1,c2,x,y = map(mpf, [1,0.5,0.25,1.125,1.5,0.0625,-0.0625])
>>> F = lambda x,y: appellf2(a,b1,b2,c1,c2,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) -
...      x*y*diff(F,(x,y),(1,1)) +
...      (c1-(a+b1+1)*x)*diff(F,(x,y),(1,0)) -
...      b1*y*diff(F,(x,y),(0,1)) -
...      a*b1*F(x,y))
0.0
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) -
...      x*y*diff(F,(x,y),(1,1)) +
...      (c2-(a+b2+1)*y)*diff(F,(x,y),(0,1)) -
...      b2*x*diff(F,(x,y),(1,0)) -
...      a*b2*F(x,y))
0.0
```

References

See references for `appellf1()` (page 808).

appellf3()

mpmath.**appellf3**(*a1, a2, b1, b2, c, x, y, **kwargs*)

Gives the Appell F3 hypergeometric function of two variables

$$F_3(a_1, a_2, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a_1)_m (a_2)_n (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m! n!}.$$

The series is generally absolutely convergent for $|x| < 1, |y| < 1$.

Examples

Evaluation for various parameters and variables:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf3(1,2,3,4,5,0.5,0.25)
2.2215577781074389381587 05
>>> appellf3(1,2,3,4,5,6,0); hyp2f1(1,3,5,6)
(-0.51895545890898612845373 89 - 0.14544410433286079807697 42j)
(-0.51895545890898612845373 89 - 0.14544410433286079807697 42j)
>>> appellf3(1,-2,-3,1,1,4,6)
-17.4
>>> appellf3(1,2,-3,1,1,4,6)
(17.78761367736773566418 25 + 19.54768762233649126154534j)
>>> appellf3(1,2,-3,1,1,6,4)
(85.02054175067929402953645 + 148.4402528821177305173599j)
>>> chop(appellf3(1+j,2,1-j,2,3,0.25,0.25))
1.7199921695452002866960 07
```

Many transformations and evaluations for special combinations of the parameters are possible, e.g.:

```
>>> a,b,c,x,y = map(mpf, [0.5,0.25,0.125,0.125,-0.125])
>>> appellf3(a,c-a,b,c-b,c,x,y)
1.0934323408960871074443 63
>>> (1-y)**(a+b-c)*hyp2f1(a,b,c,x+y-x*y)
1.0934323408960871074443 63
>>> x**2*appellf3(1,1,1,1,3,x,-x)
0.0156864627744538539094 5083
>>> polylog(2,x**2)
0.0156864627744538539094 5083
>>> a1,a2,b1,b2,c,x = map(mpf, [0.5,0.25,0.125,0.5,4.25,0.125])
>>> appellf3(a1,a2,b1,b2,c,x,1)
1.0394736170911140096947
>>> gammaprod([c,c-a2-b2],[c-a2,c-b2])*hyp3f2(a1,b1,c-a2-b2,c-a2,c-b2,x)
1.0394736170911140096947
```

The Appell F3 function satisfies a pair of partial differential equations:

```
>>> a1,a2,b1,b2,c,x,y = map(mpf, [0.5,0.25,0.125,0.5,0.625,0.0625,-0.0625])
>>> F = lambda x,y: appellf3(a1,a2,b1,b2,c,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) +
...      y*diff(F,(x,y),(1,1)) +
...     (c-(a1+b1+1)*x)*diff(F,(x,y),(1,0)) -
...     a1*b1*F(x,y))
0.0
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) +
...     x*diff(F,(x,y),(1,1)) +
...     (c-(a2+b2+1)*y)*diff(F,(x,y),(0,1)) -
...     a2*b2*F(x,y))
0.0
```

References

See references for `appellf1()` (page 808).

appellf4()

mpmath.**appellf4**(*a, b, c1, c2, x, y, **kwargs*)

Gives the Appell F4 hypergeometric function of two variables

$$F_4(a,b,c_1,c_2,x,y) = \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{(a)_{m+n}(b)_{m+n}}{(c_1)_m(c_2)_n} \frac{x^m y^n}{m!n!}.$$

The series is generally absolutely convergent for $\sqrt{|x|} + \sqrt{|y|} < 1$.

Examples

Evaluation for various parameters and arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf4(1,1,2,2,0.25,0.125)
1.286182069079718313546608
>>> appellf4(-2,-3,4,5,4,5)
34.8
>>> appellf4(5,4,2,3,0.25j,-0.125j)
(-0.2585967215437846642163352 + 2.436102233553582711818743j)
```

Reduction to ${}_2F_1$ in a special case:

```
>>> a,b,c,x,y = map(mpf, [0.5,0.25,0.125,0.125,-0.125])
>>> appellf4(a,b,c,a+b-c+1,x*(1-y),y*(1-x))
1.129143488466850868248364
>>> hyp2f1(a,b,c,x)*hyp2f1(a,b,a+b-c+1,y)
1.129143488466850868248364
```

A system of partial differential equations satisfied by F4:

```
>>> a,b,c1,c2,x,y = map(mpf, [1,0.5,0.25,1.125,0.0625,-0.0625])
>>> F = lambda x,y: appellf4(a,b,c1,c2,x,y)
>>> chop(x*(1-x)*diff(F,(x,y),(2,0)) -
...      y**2*diff(F,(x,y),(0,2)) -
...      2*x*y*diff(F,(x,y),(1,1)) +
...      (c1-(a+b+1)*x)*diff(F,(x,y),(1,0)) -
...      ((a+b+1)*y)*diff(F,(x,y),(0,1)) -
...      a*b*F(x,y))
0.0
>>> chop(y*(1-y)*diff(F,(x,y),(0,2)) -
...      x**2*diff(F,(x,y),(2,0)) -
...      2*x*y*diff(F,(x,y),(1,1)) +
...      (c2-(a+b+1)*y)*diff(F,(x,y),(0,1)) -
...      ((a+b+1)*x)*diff(F,(x,y),(1,0)) -
...      a*b*F(x,y))
0.0
```

References

See references for `appellf1()` (page 808).

Elliptic functions

Elliptic functions historically comprise the elliptic integrals and their inverses, and originate from the problem of computing the arc length of an ellipse. From a more modern point of view, an elliptic function is defined as a doubly periodic function, i.e. a function which satisfies

$$f(z + 2\omega_1) = f(z + 2\omega_2) = f(z)$$

for some half-periods ω_1, ω_2 with $\mathrm{Im}[\omega_1/\omega_2] > 0$. The canonical elliptic functions are the Jacobi elliptic functions. More broadly, this section includes quasi-doubly periodic functions (such as the Jacobi theta functions) and other functions useful in the study of elliptic functions.

Many different conventions for the arguments of elliptic functions are in use. It is even standard to use different parameterizations for different functions in the same text or software (and mpmath is no exception). The usual parameters are the elliptic nome q, which usually must satisfy $|q| < 1$; the elliptic parameter m (an arbitrary complex number); the elliptic modulus k (an arbitrary complex number); and the half-period ratio τ, which usually must satisfy $\mathrm{Im}[\tau] > 0$. These quantities can be expressed in terms of each other using the following relations:

$$m = k^2$$

$$\tau = -i\frac{K(1-m)}{K(m)}$$

$$q = e^{i\pi\tau}$$

$$k = \frac{\vartheta_2^4(q)}{\vartheta_3^4(q)}$$

In addition, an alternative definition is used for the nome in number theory, which we here denote by q-bar:

$$\bar{q} = q^2 = e^{2i\pi\tau}$$

For convenience, mpmath provides functions to convert between the various parameters (`qfrom()` (page 814), `mfrom()` (page 815), `kfrom()` (page 815), `taufrom()` (page 816), `qbarfrom()` (page 815)).

References

1. [AbramowitzStegun] (page 1625)
2. [WhittakerWatson] (page 1626)

Elliptic arguments

qfrom()

mpmath.**qfrom**(**kwargs)

Returns the elliptic nome q, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qfrom(q=0.25)
0.25
>>> qfrom(m=mfrom(q=0.25))
0.25
>>> qfrom(k=kfrom(q=0.25))
0.25
>>> qfrom(tau=taufrom(q=0.25))
(0.25 + 0.0j)
>>> qfrom(qbar=qbarfrom(q=0.25))
0.25
```

qbarfrom()

mpmath.**qbarfrom**(**kwargs)

Returns the number-theoretic nome $\bar{q}$, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qbarfrom(qbar=0.25)
0.25
>>> qbarfrom(q=qfrom(qbar=0.25))
0.25
>>> qbarfrom(m=extraprec(20)(mfrom)(qbar=0.25))  # ill-conditioned
0.25
>>> qbarfrom(k=extraprec(20)(kfrom)(qbar=0.25))  # ill-conditioned
0.25
>>> qbarfrom(tau=taufrom(qbar=0.25))
(0.25 + 0.0j)
```

mfrom()

mpmath.**mfrom**(**kwargs)

Returns the elliptic parameter m, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> mfrom(m=0.25)
0.25
>>> mfrom(q=qfrom(m=0.25))
0.25
>>> mfrom(k=kfrom(m=0.25))
0.25
>>> mfrom(tau=taufrom(m=0.25))
(0.25 + 0.0j)
>>> mfrom(qbar=qbarfrom(m=0.25))
0.25
```

As $q \to 1$ and $q \to -1$, m rapidly approaches 1 and $-\infty$ respectively:

```
>>> mfrom(q=0.75)
0.9999999999999798332943533
>>> mfrom(q=-0.75)
-49586681013729.32611558353
>>> mfrom(q=1)
1.0
>>> mfrom(q=-1)
-inf
```

The inverse nome as a function of q has an integer Taylor series expansion:

```
>>> taylor(lambda q: mfrom(q), 0, 7)
[0.0, 16.0, -128.0, 704.0, -3072.0, 11488.0, -38400.0, 117632.0]
```

kfrom()

mpmath.**kfrom**(**kwargs)

Returns the elliptic modulus k, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
```

```
>>> kfrom(k=0.25)
0.25
>>> kfrom(m=mfrom(k=0.25))
0.25
>>> kfrom(q=qfrom(k=0.25))
0.25
>>> kfrom(tau=taufrom(k=0.25))
(0.25 + 0.0j)
>>> kfrom(qbar=qbarfrom(k=0.25))
0.25
```

As $q \to 1$ and $q \to -1$, k rapidly approaches 1 and $i\infty$ respectively:

```
>>> kfrom(q=0.75)
0.999999999999999899166471767
>>> kfrom(q=-0.75)
(0.0 + 7041781.096692038332790615j)
>>> kfrom(q=1)
1
>>> kfrom(q=-1)
(0.0 + +infj)
```

taufrom()

mpmath.**taufrom**(**kwargs)

Returns the elliptic half-period ratio τ, given any of $q, m, k, \tau, \bar{q}$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> taufrom(tau=0.5j)
(0.0 + 0.5j)
>>> taufrom(q=qfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(m=mfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(k=kfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(qbar=qbarfrom(tau=0.5j))
(0.0 + 0.5j)
```

Legendre elliptic integrals

ellipk()

mpmath.**ellipk**(*m*, **kwargs)

Evaluates the complete elliptic integral of the first kind, $K(m)$, defined by

$$K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \frac{\pi}{2} \, {}_2F_1\left(\frac{1}{2}, \frac{1}{2}, 1, m\right).$$

Note that the argument is the parameter $m = k^2$, not the modulus k which is sometimes used.

Plots

```
# Complete elliptic integrals K(m) and E(m)
plot([ellipk, ellipe], [-2,1], [0,3], points=600)
```

Examples

Values and limits include:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipk(0)
1.5707963267948966192313 22
>>> ellipk(inf)
(0.0 + 0.0j)
>>> ellipk(-inf)
0.0
>>> ellipk(1)
+inf
>>> ellipk(-1)
1.3110287771460599052324 2
>>> ellipk(2)
(1.3110287771460599052324 2 - 1.3110287771460599052324 2j)
```

Verifying the defining integral and hypergeometric representation:

```
>>> ellipk(0.5)
1.8540746773013719184338 5
>>> quad(lambda t: (1-0.5*sin(t)**2)**-0.5, [0, pi/2])
1.8540746773013719184338 5
>>> pi/2*hyp2f1(0.5,0.5,1,0.5)
```

```
1.8540746773013719184338 5
```

Evaluation is supported for arbitrary complex m:

```
>>> ellipk(3+4j)
(0.91119556380496500865621 71 + 0.63133428324134524388450 91j)
```

A definite integral:

```
>>> quad(ellipk, [0, 1])
2.0
```

ellipf()

mpmath.**ellipf**(*phi*, *m*)

Evaluates the Legendre incomplete elliptic integral of the first kind

$$F(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

or equivalently

$$F(\phi, m) = \int_0^{\sin z} \frac{dt}{\left(\sqrt{1 - t^2}\right)\left(\sqrt{1 - mt^2}\right)}.$$

The function reduces to a complete elliptic integral of the first kind (see `ellipk()` (page 816)) when $\phi = \frac{\pi}{2}$; that is,

$$F\left(\frac{\pi}{2}, m\right) = K(m).$$

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside $-\pi/2 \le \Re(z) \le \pi/2$, the function extends quasi-periodically as

$$F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}.$$

Plots

```
# Elliptic integral F(z,m) for some different m
f1 = lambda z: ellipf(z,-1)
f2 = lambda z: ellipf(z,-0.5)
f3 = lambda z: ellipf(z,0)
f4 = lambda z: ellipf(z,0.5)
f5 = lambda z: ellipf(z,1)
plot([f1,f2,f3,f4,f5], [0,pi], [0,4])
```

Examples

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipf(0,1)
0.0
>>> ellipf(0,0)
0.0
>>> ellipf(1,0); ellipf(2+3j,0)
1.0
(2.0 + 3.0j)
>>> ellipf(1,1); log(sec(1)+tan(1))
1.2261911708835170708130 61
1.2261911708835170708130 61
>>> ellipf(pi/2, -0.5); ellipk(-0.5)
1.4157372084259561988921 66
1.4157372084259561988921 66
>>> ellipf(pi/2+eps, 1); ellipf(-pi/2-eps, 1)
+inf
+inf
>>> ellipf(1.5, 1)
3.3406775427983110033208 13
```

Comparing with numerical integration:

```
>>> z,m = 0.5, 1.25
>>> ellipf(z,m)
0.52872192022063278729782 55
>>> quad(lambda t: (1-m*sin(t)**2)**(-0.5), [0,z])
0.52872192022063278729782 55
```

The arguments may be complex numbers:

```
>>> ellipf(3j, 0.5)
(0.0 + 1.7136024078415902348041 43j)
>>> ellipf(3+4j, 5-6j)
(1.2691312419503513233057 41 - 0.3561052815014558335412538j)
>>> z,m = 2+3j, 1.25
>>> k = 1011
>>> ellipf(z+pi*k,m); ellipf(z,m) + 2*k*ellipk(m)
(4086.184383622179764082821 - 3003.003538923749396546871j)
(4086.184383622179764082821 - 3003.003538923749396546871j)
```

For $|\Re(z)| < \pi/2$, the function can be expressed as a hypergeometric series of two variables (see `appellf1()` (page 808)):

```
>>> z,m = 0.5, 0.25
>>> ellipf(z,m)
0.5050887275786480788831083
>>> sin(z)*appellf1(0.5,0.5,0.5,1.5,sin(z)**2,m*sin(z)**2)
0.5050887275786480788831083
```

ellipe()

mpmath.**ellipe**(**args*)

Called with a single argument m, evaluates the Legendre complete elliptic integral of the second kind, $E(m)$, defined by

$$E(m) = \int_0^{\pi/2} \sqrt{1 - m \sin^2 t}\, dt \;=\; \frac{\pi}{2}\, {}_2F_1\left(\frac{1}{2}, -\frac{1}{2}, 1, m\right).$$

Called with two arguments ϕ, m, evaluates the incomplete elliptic integral of the second kind

$$E(\phi, m) = \int_0^{\phi} \sqrt{1 - m \sin^2 t}\, dt = \int_0^{\sin z} \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}}\, dt.$$

The incomplete integral reduces to a complete integral when $\phi = \frac{\pi}{2}$; that is,

$$E\left(\frac{\pi}{2}, m\right) = E(m).$$

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside $-\pi/2 \le \Re(z) \le \pi/2$, the function extends quasi-periodically as

$$E(\phi + n\pi, m) = 2nE(m) + F(\phi, m), n \in \mathbb{Z}.$$

Plots

```
# Elliptic integral E(z,m) for some different m
f1 = lambda z: ellipe(z,-2)
f2 = lambda z: ellipe(z,-1)
f3 = lambda z: ellipe(z,0)
f4 = lambda z: ellipe(z,1)
f5 = lambda z: ellipe(z,2)
plot([f1,f2,f3,f4,f5], [0,pi], [0,4])
```

Examples for the complete integral

Basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipe(0)
1.570796326794896619231322
>>> ellipe(1)
1.0
>>> ellipe(-1)
1.910098894513856008952381
>>> ellipe(2)
(0.5990701173677961037199612 + 0.5990701173677961037199612j)
>>> ellipe(inf)
(0.0 + +infj)
>>> ellipe(-inf)
+inf
```

Verifying the defining integral and hypergeometric representation:

```
>>> ellipe(0.5)
1.35064388104767550252017 5
>>> quad(lambda t: sqrt(1-0.5*sin(t)**2), [0, pi/2])
1.35064388104767550252017 5
>>> pi/2*hyp2f1(0.5,-0.5,1,0.5)
1.35064388104767550252017 5
```

Evaluation is supported for arbitrary complex m:

```
>>> ellipe(0.5+0.25j)
(1.360868682163129682716687 - 0.12387334425617868435573 15j)
>>> ellipe(3+4j)
(1.49955352093334695433361 2 - 1.57787900791275827453330 9j)
```

A definite integral:

```
>>> quad(ellipe, [0,1])
1.33333333333333333333333
```

Examples for the incomplete integral

Basic values and limits:

```
>>> ellipe(0,1)
0.0
>>> ellipe(0,0)
0.0
>>> ellipe(1,0)
1.0
>>> ellipe(2+3j,0)
(2.0 + 3.0j)
>>> ellipe(1,1); sin(1)
0.841470984807896506652502 3
0.841470984807896506652502 3
>>> ellipe(pi/2, -0.5); ellipe(-0.5)
1.75177127569481786202650 2
1.75177127569481786202650 2
>>> ellipe(pi/2, 1); ellipe(-pi/2, 1)
1.0
-1.0
>>> ellipe(1.5, 1)
0.997494986604054430941723 4
```

Comparing with numerical integration:

```
>>> z,m = 0.5, 1.25
>>> ellipe(z,m)
0.474015218265262839426444 9
>>> quad(lambda t: sqrt(1-m*sin(t)**2), [0,z])
0.474015218265262839426444 9
```

The arguments may be complex numbers:

```
>>> ellipe(3j, 0.5)
(0.0 + 7.55199123489037187350210 5j)
>>> ellipe(3+4j, 5-6j)
(24.1529902257422050242446 6 + 75.2503670480325997418156j)
>>> k = 35
>>> z,m = 2+3j, 1.25
>>> ellipe(z+pi*k,m); ellipe(z,m) + 2*k*ellipe(m)
```

```
(48.30138799412005235090766 + 17.47255216721987688224357j)
(48.30138799412005235090766 + 17.47255216721987688224357j)
```

For $|\Re(z)| < \pi/2$, the function can be expressed as a hypergeometric series of two variables (see `appellf1()` (page 808)):

```
>>> z,m = 0.5, 0.25
>>> ellipe(z,m)
0.495001703016415192887037 5
>>> sin(z)*appellf1(0.5,0.5,-0.5,1.5,sin(z)**2,m*sin(z)**2)
0.495001703016415192887037 6
```

ellippi()

mpmath.**ellippi**(**args*)

Called with three arguments n, ϕ, m, evaluates the Legendre incomplete elliptic integral of the third kind

$$\Pi(n;\phi,m) = \int_0^{\phi} \frac{dt}{(1-n\sin^2 t)\sqrt{1-m\sin^2 t}} = \int_0^{\sin\phi} \frac{dt}{(1-nt^2)\sqrt{1-t^2}\sqrt{1-mt^2}}.$$

Called with two arguments n, m, evaluates the complete elliptic integral of the third kind $\Pi(n,m) = \Pi(n;\frac{\pi}{2},m)$.

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside $-\pi/2 \le \Re(z) \le \pi/2$, the function extends quasi-periodically as

$$\Pi(n,\phi+k\pi,m) = 2k\Pi(n,m) + \Pi(n,\phi,m), k \in \mathbb{Z}.$$

Plots

```
# Elliptic integral Pi(n,z,m) for some different n, m
f1 = lambda z: ellippi(0.9,z,0.9)
f2 = lambda z: ellippi(0.5,z,0.5)
f3 = lambda z: ellippi(-2,z,-0.9)
f4 = lambda z: ellippi(-0.5,z,0.5)
f5 = lambda z: ellippi(-1,z,0.5)
plot([f1,f2,f3,f4,f5], [0,pi], [0,4])
```

Examples for the complete integral

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellippi(0,-5); ellipk(-5)
0.9555039270640439337379334
0.9555039270640439337379334
>>> ellippi(inf,2)
0.0
>>> ellippi(2,inf)
0.0
>>> abs(ellippi(1,5))
+inf
>>> abs(ellippi(0.25,1))
+inf
```

Evaluation in terms of simpler functions:

```
>>> ellippi(0.25,0.25); ellipe(0.25)/(1-0.25)
1.956616279119236207279727
1.956616279119236207279727
>>> ellippi(3,0); pi/(2*sqrt(-2))
(0.0 - 1.110720734539591561753 97j)
(0.0 - 1.110720734539591561753 97j)
>>> ellippi(-3,0); pi/(2*sqrt(4))
0.7853981633974483096156609
0.7853981633974483096156609
```

Examples for the incomplete integral

Basic values and limits:

```
>>> ellippi(0.25,-0.5); ellippi(0.25,pi/2,-0.5)
1.6229447609547416037105 55
1.6229447609547416037105 55
>>> ellippi(1,0,1)
0.0
>>> ellippi(inf,0,1)
0.0
>>> ellippi(0,0.25,0.5); ellipf(0.25,0.5)
0.2513040086544925794134591
0.2513040086544925794134591
>>> ellippi(1,1,1); (log(sec(1)+tan(1))+sec(1)*tan(1))/2
2.0543329332562486686924 52
2.0543329332562486686924 52
>>> ellippi(0.25, 53*pi/2, 0.75); 53*ellippi(0.25,0.75)
135.24086875789084075505 8
135.24086875789084075505 8
>>> ellippi(0.5,pi/4,0.5); 2*ellipe(pi/4,0.5)-1/sqrt(3)
0.9190227391656969903987269
0.9190227391656969903987269
```

Complex arguments are supported:

```
>>> ellippi(0.5, 5+6j-2*pi, -7-8j)
(-0.36128566200767476604101 67 + 0.52177353399848078297558 15j)
```

Carlson symmetric elliptic integrals

elliprf()

mpmath.**elliprf**(*x, y, z*)

Evaluates the Carlson symmetric elliptic integral of the first kind

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}$$

which is defined for $x, y, z \notin (-\infty, 0)$, and with at most one of x, y, z being zero.

For real $x, y, z \geq 0$, the principal square root is taken in the integrand. For complex x, y, z, the principal square root is taken as $t \to \infty$ and as $t \to 0$ non-principal branches are chosen as necessary so as to make the integrand continuous.

Examples

Some basic values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprf(0,1,1); pi/2
1.5707963267948966192313 22
1.5707963267948966192313 22
>>> elliprf(0,1,inf)
0.0
>>> elliprf(1,1,1)
1.0
>>> elliprf(2,2,2)**2
```

```
0.5
>>> elliprf(1,0,0); elliprf(0,0,1); elliprf(0,1,0); elliprf(0,0,0)
+inf
+inf
+inf
+inf
```

Representing complete elliptic integrals in terms of R_F:

```
>>> m = mpf(0.75)
>>> ellipk(m); elliprf(0,1-m,1)
2.156515647499643235438675
2.156515647499643235438675
>>> ellipe(m); elliprf(0,1-m,1)-m*elliprd(0,1-m,1)/3
1.211056027568459524803563
1.211056027568459524803563
```

Some symmetries and argument transformations:

```
>>> x,y,z = 2,3,4
>>> elliprf(x,y,z); elliprf(y,x,z); elliprf(z,y,x)
0.584082841677151706692849 2
0.584082841677151706692849 2
0.584082841677151706692849 2
>>> k = mpf(100000)
>>> elliprf(k*x,k*y,k*z); k**(-0.5) * elliprf(x,y,z)
0.00184703212192332125321928 4
0.00184703212192332125321928 4
>>> l = sqrt(x*y) + sqrt(y*z) + sqrt(z*x)
>>> elliprf(x,y,z); 2*elliprf(x+l,y+l,z+l)
0.584082841677151706692849 2
0.584082841677151706692849 2
>>> elliprf((x+l)/4,(y+l)/4,(z+l)/4)
0.584082841677151706692849 2
```

Comparing with numerical integration:

```
>>> x,y,z = 2,3,4
>>> elliprf(x,y,z)
0.584082841677151706692849 2
>>> f = lambda t: 0.5*((t+x)*(t+y)*(t+z))**(-0.5)
>>> q = extradps(25)(quad)
>>> q(f, [0,inf])
0.584082841677151706692849 2
```

With the following arguments, the square root in the integrand becomes discontinuous at $t = 1/2$ if the principal branch is used. To obtain the right value, $-\sqrt{r}$ must be taken instead of $\sqrt{r}$ on $t \in (0, 1/2)$:

```
>>> x,y,z = j-1,j,0
>>> elliprf(x,y,z)
(0.796125865842339132930569 4 - 1.21385666983649598643009 4j)
>>> -q(f, [0,0.5]) + q(f, [0.5,inf])
(0.796125865842339132930569 4 - 1.21385666983649598643009 4j)
```

The so-called *first lemniscate constant*, a transcendental number:

```
>>> elliprf(0,1,2)
1.311028777146059905232 42
>>> extradps(25)(quad)(lambda t: 1/sqrt(1-t**4), [0,1])
```

```
1.3110287771460599052324 2
>>> gamma('1/4')**2/(4*sqrt(2*pi))
1.3110287771460599052324 2
```

References

1. [Carlson] (page 1625)
2. [DLMF] (page 1625) Chapter 19. Elliptic Integrals

elliprc()

mpmath.**elliprc**(*x*, *y*, *pv=True*)

Evaluates the degenerate Carlson symmetric elliptic integral of the first kind

$$R_C(x,y) = R_F(x,y,y) = \frac{1}{2}\int_0^\infty \frac{dt}{(t+y)\sqrt{(t+x)}}.$$

If $y \in (-\infty, 0)$, either a value defined by continuity, or with *pv=True* the Cauchy principal value, can be computed.

If $x \geq 0, y > 0$, the value can be expressed in terms of elementary functions as

$$R_C(x,y) = \begin{cases} \frac{1}{\sqrt{y-x}}\cos^{-1}\left(\sqrt{\frac{x}{y}}\right), & x < y \\ \frac{1}{\sqrt{y}}, & x = y \\ \frac{1}{\sqrt{x-y}}\cosh^{-1}\left(\sqrt{\frac{x}{y}}\right), & x > y \end{cases}.$$

Examples

Some special values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprc(1,2)*4; elliprc(0,1)*2; +pi
3.141592653589793238462643
3.141592653589793238462643
3.141592653589793238462643
>>> elliprc(1,0)
+inf
>>> elliprc(5,5)**2
0.2
>>> elliprc(1,inf); elliprc(inf,1); elliprc(inf,inf)
0.0
0.0
0.0
```

Comparing with the elementary closed-form solution:

```
>>> elliprc('1/3', '1/5'); sqrt(7.5)*acosh(sqrt('5/3'))
2.041630778983498390751238
2.041630778983498390751238
>>> elliprc('1/5', '1/3'); sqrt(7.5)*acos(sqrt('3/5'))
1.875180765206547065111085
1.875180765206547065111085
```

Comparing with numerical integration:

```
>>> q = extradps(25)(quad)
>>> elliprc(2, -3, pv=True)
0.3333969101113672670749334
>>> elliprc(2, -3, pv=False)
(0.3333969101113672670749334 + 0.7024814731040726393156375j)
>>> 0.5*q(lambda t: 1/(sqrt(t+2)*(t-3)), [0,3-j,6,inf])
(0.3333969101113672670749334 + 0.7024814731040726393156375j)
```

elliprj()

mpmath.**elliprj**(*x, y, z, p*)

Evaluates the Carlson symmetric elliptic integral of the third kind

$$R_J(x,y,z,p) = \frac{3}{2}\int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}}.$$

Like `elliprf()` (page 825), the branch of the square root in the integrand is defined so as to be continuous along the path of integration for complex values of the arguments.

Examples

Some values and limits:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprj(1,1,1,1)
1.0
>>> elliprj(2,2,2,2); 1/(2*sqrt(2))
0.3535533905932737622004222
0.3535533905932737622004222
>>> elliprj(0,1,2,2)
1.067937989667395702268688
>>> 3*(2*gamma('5/4')**2-pi**2/gamma('1/4')**2)/(sqrt(2*pi))
1.067937989667395702268688
>>> elliprj(0,1,1,2); 3*pi*(2-sqrt(2))/4
1.380226776765915172432054
1.380226776765915172432054
>>> elliprj(1,3,2,0); elliprj(0,1,1,0); elliprj(0,0,0,0)
+inf
+inf
+inf
>>> elliprj(1,inf,1,0); elliprj(1,1,1,inf)
0.0
0.0
>>> chop(elliprj(1+j, 1-j, 1, 1))
0.8505007163686739432927844
```

Scale transformation:

```
>>> x,y,z,p = 2,3,4,5
>>> k = mpf(100000)
>>> elliprj(k*x,k*y,k*z,k*p); k**(-1.5)*elliprj(x,y,z,p)
4.521291677592745527851168e-9
4.521291677592745527851168e-9
```

Comparing with numerical integration:

```
>>> elliprj(1,2,3,4)
0.2398480997495677621758617
```

```
>>> f = lambda t: 1/((t+4)*sqrt((t+1)*(t+2)*(t+3)))
>>> 1.5*quad(f, [0,inf])
0.23984809974957677621758617
>>> elliprj(1,2+1j,3,4-2j)
(0.21688890601463349873995 2 + 0.040819126273666733323695 12j)
>>> f = lambda t: 1/((t+4-2j)*sqrt((t+1)*(t+2+1j)*(t+3)))
>>> 1.5*quad(f, [0,inf])
(0.216888906014633498739952 + 0.040819126273666733323695 11j)
```

elliprd()

mpmath.**elliprd**(*x, y, z*)

Evaluates the degenerate Carlson symmetric elliptic integral of the third kind or Carlson elliptic integral of the second kind $R_D(x,y,z) = R_J(x,y,z,z)$.

See elliprj() (page 828) for additional information.

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprd(1,2,3)
0.2904602810289906442326534
>>> elliprj(1,2,3,3)
0.2904602810289906442326534
```

The so-called *second lemniscate constant,* a transcendental number:

```
>>> elliprd(0,2,1)/3
0.5990701173677961037199612
>>> extradps(25)(quad)(lambda t: t**2/sqrt(1-t**4), [0,1])
0.5990701173677961037199612
>>> gamma('3/4')**2/sqrt(2*pi)
0.5990701173677961037199612
```

elliprg()

mpmath.**elliprg**(*x, y, z*)

Evaluates the Carlson completely symmetric elliptic integral of the second kind

$$R_G(x,y,z) = \frac{1}{4}\int_0^\infty \frac{t}{\sqrt{(t+x)(t+y)(t+z)}}\left(\frac{x}{t+x}+\frac{y}{t+y}+\frac{z}{t+z}\right)dt.$$

Examples

Evaluation for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprg(0,1,1)*4; +pi
3.141592653589793238462643
3.141592653589793238462643
>>> elliprg(0,0.5,1)
0.6753219405238377512600874
>>> chop(elliprg(1+j, 1-j, 2))
1.172431327676416604532822
```

A double integral that can be evaluated in terms of R_G:

```
>>> x,y,z = 2,3,4
>>> def f(t,u):
...     st = fp.sin(t); ct = fp.cos(t)
...     su = fp.sin(u); cu = fp.cos(u)
...     return (x*(st*cu)**2 + y*(st*su)**2 + z*ct**2)**0.5 * st
...
>>> nprint(mpf(fp.quad(f, [0,fp.pi], [0,2*fp.pi])/(4*fp.pi)), 13)
1.725503028069
>>> nprint(elliprg(x,y,z), 13)
1.725503028069
```

Jacobi theta functions

jtheta()

mpmath.**jtheta**(*n, z, q, derivative=0*)

Computes the Jacobi theta function $\vartheta_n(z,q)$, where $n = 1,2,3,4$, defined by the infinite series:

$$\vartheta_1(z,q) = 2q^{1/4}\sum_{n=0}^{\infty}(-1)^n q^{n^2+n}\sin((2n+1)z)$$

$$\vartheta_2(z,q) = 2q^{1/4}\sum_{n=0}^{\infty}q^{n^2+n}\cos((2n+1)z)$$

$$\vartheta_3(z,q) = 1 + 2\sum_{n=1}^{\infty}q^{n^2}\cos(2nz)$$

$$\vartheta_4(z,q) = 1 + 2\sum_{n=1}^{\infty}(-q)^{n^2}\cos(2nz)$$

The theta functions are functions of two variables:

- z is the *argument*, an arbitrary real or complex number
- q is the *nome*, which must be a real or complex number in the unit disk (i.e. $|q| < 1$). For $|q| \ll 1$, the series converge very quickly, so the Jacobi theta functions can efficiently be evaluated to high precision.

The compact notations $\vartheta_n(q) = \vartheta_n(0,q)$ and $\vartheta_n = \vartheta_n(0,q)$ are also frequently encountered. Finally, Jacobi theta functions are frequently considered as functions of the half-period ratio τ and then usually denoted by $\vartheta_n(z|\tau)$.

Optionally, `jtheta(n, z, q, derivative=d)` with $d > 0$ computes a d-th derivative with respect to z.

Examples and basic properties

Considered as functions of z, the Jacobi theta functions may be viewed as generalizations of the ordinary trigonometric functions cos and sin. They are periodic functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> jtheta(1, 0.25, '0.2')
0.2945120798627300045053104
>>> jtheta(1, 0.25 + 2*pi, '0.2')
0.2945120798627300045053104
```

Indeed, the series defining the theta functions are essentially trigonometric Fourier series. The coefficients can be retrieved using fourier() (page 913):

```
>>> mp.dps = 10
>>> nprint(fourier(lambda x: jtheta(2, x, 0.5), [-pi, pi], 4))
([0.0, 1.68179, 0.0, 0.420448, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0])
```

The Jacobi theta functions are also so-called quasiperiodic functions of z and τ, meaning that for fixed τ, $\vartheta_n(z,q)$ and $\vartheta_n(z+\pi\tau,q)$ are the same except for an exponential factor:

```
>>> mp.dps = 25
>>> tau = 3*j/10
>>> q = exp(pi*j*tau)
>>> z = 10
>>> jtheta(4, z+tau*pi, q)
(-0.6824202807860346875205 68 + 1.526683999721399103332021j)
>>> -exp(-2*j*z)/q * jtheta(4, z, q)
(-0.6824202807860346875205 68 + 1.526683999721399103332021j)
```

The Jacobi theta functions satisfy a huge number of other functional equations, such as the following identity (valid for any q):

```
>>> q = mpf(3)/10
>>> jtheta(3,0,q)**4
6.82374408935276330513742 7
>>> jtheta(2,0,q)**4 + jtheta(4,0,q)**4
6.82374408935276330513742 7
```

Extensive listings of identities satisfied by the Jacobi theta functions can be found in standard reference works.

The Jacobi theta functions are related to the gamma function for special arguments:

```
>>> jtheta(3, 0, exp(-pi))
1.0864348112133080145753 16
>>> pi**(1/4.) / gamma(3/4.)
1.0864348112133080145753 16
```

jtheta() (page 830) supports arbitrary precision evaluation and complex arguments:

```
>>> mp.dps = 50
>>> jtheta(4, sqrt(2), 0.5)
2.0549510717571539127004115835148878097035750653737
>>> mp.dps = 25
>>> jtheta(4, 1+2j, (1+j)/5)
(7.1803317601468059263566 34 - 1.6342928581191624173016 83j)
```

Evaluation of derivatives:

```
>>> mp.dps = 25
>>> jtheta(1, 7, 0.25, 1); diff(lambda z: jtheta(1, z, 0.25), 7)
1.2098571928444753886372 36
1.2098571928444753886372 36
>>> jtheta(1, 7, 0.25, 2); diff(lambda z: jtheta(1, z, 0.25), 7, 2)
-0.2598718791650217206533052
-0.2598718791650217206533052
>>> jtheta(2, 7, 0.25, 1); diff(lambda z: jtheta(2, z, 0.25), 7)
-1.1502314370702596444614 74
-1.1502314370702596444614 74
>>> jtheta(2, 7, 0.25, 2); diff(lambda z: jtheta(2, z, 0.25), 7, 2)
```

```
-0.622663699004377744589811 4
-0.622663699004377744589811 4
>>> jtheta(3, 7, 0.25, 1); diff(lambda z: jtheta(3, z, 0.25), 7)
-0.999031204609663431658788 2
-0.999031204609663431658788 2
>>> jtheta(3, 7, 0.25, 2); diff(lambda z: jtheta(3, z, 0.25), 7, 2)
-0.153038869306633493615117 4
-0.153038869306633493615117 4
>>> jtheta(4, 7, 0.25, 1); diff(lambda z: jtheta(4, z, 0.25), 7)
0.982099596726279394357113 9
0.982099596726279394357113 9
>>> jtheta(4, 7, 0.25, 2); diff(lambda z: jtheta(4, z, 0.25), 7, 2)
0.393690285029143708166775 5
0.393690285029143708166775 5
```

Possible issues

For $|q| \geq 1$ or $\Im(\tau) \leq 0$, `jtheta()` (page 830) raises `ValueError`. This exception is also raised for $|q|$ extremely close to 1 (or equivalently τ very close to 0), since the series would converge too slowly:

```
>>> jtheta(1, 10, 0.99999999 * exp(0.5*j))
Traceback (most recent call last):
  ...
ValueError: abs(q) > THETA_Q_LIM = 1.000000
```

Jacobi elliptic functions

ellipfun()

mpmath.**ellipfun**(*kind*, *u=None*, *m=None*, *q=None*, *k=None*, *tau=None*)

Computes any of the Jacobi elliptic functions, defined in terms of Jacobi theta functions as

$$\operatorname{sn}(u,m) = \frac{\vartheta_3(0,q)}{\vartheta_2(0,q)}\frac{\vartheta_1(t,q)}{\vartheta_4(t,q)}$$
$$\operatorname{cn}(u,m) = \frac{\vartheta_4(0,q)}{\vartheta_2(0,q)}\frac{\vartheta_2(t,q)}{\vartheta_4(t,q)}$$
$$\operatorname{dn}(u,m) = \frac{\vartheta_4(0,q)}{\vartheta_3(0,q)}\frac{\vartheta_3(t,q)}{\vartheta_4(t,q)},$$

or more generally computes a ratio of two such functions. Here $t = u/\vartheta_3(0,q)^2$, and $q = q(m)$ denotes the nome (see `nome()`). Optionally, you can specify the nome directly instead of m by passing `q=<value>`, or you can directly specify the elliptic parameter k with `k=<value>`.

The first argument should be a two-character string specifying the function using any combination of `'s'`, `'c'`, `'d'`, `'n'`. These letters respectively denote the basic functions $\operatorname{sn}(u,m)$, $\operatorname{cn}(u,m)$, $\operatorname{dn}(u,m)$, and 1. The identifier specifies the ratio of two such functions. For example, `'ns'` identifies the function

$$\operatorname{ns}(u,m) = \frac{1}{\operatorname{sn}(u,m)}$$

and `'cd'` identifies the function

$$\operatorname{cd}(u,m) = \frac{\operatorname{cn}(u,m)}{\operatorname{dn}(u,m)}.$$

If called with only the first argument, a function object evaluating the chosen function for given arguments is returned.

Examples

Basic evaluation:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipfun('cd', 3.5, 0.5)
-0.9891101840595543931308394
>>> ellipfun('cd', 3.5, q=0.25)
0.07111979240214668158441418
```

The sn-function is doubly periodic in the complex plane with periods $4K(m)$ and $2iK(1-m)$ (see `ellipk()` (page 816)):

```
>>> sn = ellipfun('sn')
>>> sn(2, 0.25)
0.9628981775982774425751399
>>> sn(2+4*ellipk(0.25), 0.25)
0.9628981775982774425751399
>>> chop(sn(2+2*j*ellipk(1-0.25), 0.25))
0.9628981775982774425751399
```

The cn-function is doubly periodic with periods $4K(m)$ and $4iK(1-m)$:

```
>>> cn = ellipfun('cn')
>>> cn(2, 0.25)
-0.2698649654510865792581416
>>> cn(2+4*ellipk(0.25), 0.25)
-0.2698649654510865792581416
>>> chop(cn(2+4*j*ellipk(1-0.25), 0.25))
-0.2698649654510865792581416
```

The dn-function is doubly periodic with periods $2K(m)$ and $4iK(1-m)$:

```
>>> dn = ellipfun('dn')
>>> dn(2, 0.25)
0.8764740583123262286931578
>>> dn(2+2*ellipk(0.25), 0.25)
0.8764740583123262286931578
>>> chop(dn(2+4*j*ellipk(1-0.25), 0.25))
0.8764740583123262286931578
```

Modular functions

kleinj()

mpmath.**kleinj**(*tau=None, **kwargs*)

Evaluates the Klein j-invariant, which is a modular function defined for τ in the upper half-plane as

$$J(\tau) = \frac{g_2^3(\tau)}{g_2^3(\tau) - 27g_3^2(\tau)}$$

where g_2 and g_3 are the modular invariants of the Weierstrass elliptic function,

$$g_2(\tau) = 60 \sum_{(m,n)\in\mathbb{Z}^2\setminus(0,0)} (m\tau + n)^{-4}$$
$$g_3(\tau) = 140 \sum_{(m,n)\in\mathbb{Z}^2\setminus(0,0)} (m\tau + n)^{-6}.$$

An alternative, common notation is that of the j-function $j(\tau) = 1728J(\tau)$.

Plots

```
# Klein J-function as function of the number-theoretic nome
fp.cplot(lambda q: fp.kleinj(qbar=q), [-1,1], [-1,1], points=50000)
```

```
# Klein J-function as function of the half-period ratio
fp.cplot(lambda t: fp.kleinj(tau=t), [-1,2], [0,1.5], points=50000)
```

Examples

Verifying the functional equation $J(\tau) = J(\tau + 1) = J(-\tau^{-1})$:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> tau = 0.625+0.75*j
>>> tau = 0.625+0.75*j
>>> kleinj(tau)
(-0.1507492166511182267125242 + 0.07595948379084571927228948j)
>>> kleinj(tau+1)
(-0.1507492166511182267125242 + 0.07595948379084571927228948j)
>>> kleinj(-1/tau)
(-0.1507492166511182267125242 + 0.07595948379084571927228946j)
```

The j-function has a famous Laurent series expansion in terms of the nome $\bar{q}$, $j(\tau) = \bar{q}^{-1} + 744 + 196884\bar{q} + \ldots$:

```
>>> mp.dps = 15
>>> taylor(lambda q: 1728*q*kleinj(qbar=q), 0, 5, singular=True)
[1.0, 744.0, 196884.0, 21493760.0, 864299970.0, 20245856256.0]
```

The j-function admits exact evaluation at special algebraic points related to the Heegner numbers 1, 2, 3, 7, 11, 19, 43, 67, 163:

```
>>> @extraprec(10)
... def h(n):
...     v = (1+sqrt(n)*j)
...     if n > 2:
```

```
...         v *= 0.5
...     return v
...
>>> mp.dps = 25
>>> for n in [1,2,3,7,11,19,43,67,163]:
...     n, chop(1728*kleinj(h(n)))
...
(1, 1728.0)
(2, 8000.0)
(3, 0.0)
(7, -3375.0)
(11, -32768.0)
(19, -884736.0)
(43, -884736000.0)
(67, -147197952000.0)
(163, -262537412640768000.0)
```

Also at other special points, the j-function assumes explicit algebraic values, e.g.:

```
>>> chop(1728*kleinj(j*sqrt(5)))
1264538.909475140509320227
>>> identify(cbrt(_))      # note: not simplified
'((100+sqrt(13520))/2)'
>>> (50+26*sqrt(5))**3
1264538.909475140509320227
```

Zeta functions, L-series and polylogarithms

This section includes the Riemann zeta functions and associated functions pertaining to analytic number theory.

Riemann and Hurwitz zeta functions

zeta()

mpmath.**zeta**(*s*, *a=1*, *derivative=0*)

Computes the Riemann zeta function

$$\zeta(s) = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \ldots$$

or, with $a \neq 1$, the more general Hurwitz zeta function

$$\zeta(s,a) = \sum_{k=0}^{\infty} \frac{1}{(a+k)^s}.$$

Optionally, `zeta(s, a, n)` computes the n-th derivative with respect to s,

$$\zeta^{(n)}(s,a) = (-1)^n \sum_{k=0}^{\infty} \frac{\log^n(a+k)}{(a+k)^s}.$$

Although these series only converge for $\Re(s) > 1$, the Riemann and Hurwitz zeta functions are defined through analytic continuation for arbitrary complex $s \neq 1$ ($s = 1$ is a pole).

The implementation uses three algorithms: the Borwein algorithm for the Riemann zeta function when s is close to the real line; the Riemann-Siegel formula for the Riemann zeta

function when s is large imaginary, and Euler-Maclaurin summation in all other cases. The reflection formula for $\Re(s) < 0$ is implemented in some cases. The algorithm can be chosen with `method = 'borwein'`, `method='riemann-siegel'` or `method = 'euler-maclaurin'`.

The parameter a is usually a rational number $a = p/q$, and may be specified as such by passing an integer tuple (p, q). Evaluation is supported for arbitrary complex a, but may be slow and/or inaccurate when $\Re(s) < 0$ for nonrational a or when computing derivatives.

Examples

Some values of the Riemann zeta function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> zeta(2); pi**2 / 6
1.644934066848226436472415
1.644934066848226436472415
>>> zeta(0)
-0.5
>>> zeta(-1)
-0.083333333333333333333333
>>> zeta(-2)
0.0
```

For large positive s, $\zeta(s)$ rapidly approaches 1:

```
>>> zeta(50)
1.000000000000000888178421
>>> zeta(100)
1.0
>>> zeta(inf)
1.0
>>> 1-sum((zeta(k)-1)/k for k in range(2,85)); +euler
0.577215664901532860606512 1
0.577215664901532860606512 1
>>> nsum(lambda k: zeta(k)-1, [2, inf])
1.0
```

Evaluation is supported for complex s and a:

```
>>> zeta(-3+4j)
(-0.033730573388277570675846 98 + 0.277449925155709374529767 7j)
>>> zeta(2+3j, -1+j)
(389.6841230140842816370741 + 295.2674610150305334025962j)
```

The Riemann zeta function has so-called nontrivial zeros on the critical line $s = 1/2 + it$:

```
>>> findroot(zeta, 0.5+14j); zetazero(1)
(0.5 + 14.13472514173469379045725j)
(0.5 + 14.13472514173469379045725j)
>>> findroot(zeta, 0.5+21j); zetazero(2)
(0.5 + 21.02203963877155499262848j)
(0.5 + 21.02203963877155499262848j)
>>> findroot(zeta, 0.5+25j); zetazero(3)
(0.5 + 25.01085758014568876321379j)
(0.5 + 25.01085758014568876321379j)
>>> chop(zeta(zetazero(10)))
0.0
```

Evaluation on and near the critical line is supported for large heights t by means of the Riemann-Siegel formula (currently for $a = 1$, $n \leq 4$):

```
>>> zeta(0.5+100000j)
(1.07303201485775313211407 6 + 5.7808485443635039842610 41j)
>>> zeta(0.75+1000000j)
(0.95353160583751450203515 59 + 0.95259458948342730601756 51j)
>>> zeta(0.5+10000000j)
(11.4580406105770925450022 7 - 8.6434372268360217238182 15j)
>>> zeta(0.5+100000000j, derivative=1)
(51.1243310671019494268186 9 + 43.8722116787230452059941 8j)
>>> zeta(0.5+100000000j, derivative=2)
(-444.2760822795430400549229 - 896.3789978119185981665403j)
>>> zeta(0.5+100000000j, derivative=3)
(3230.7268268767042221533 9 + 14374.3695007361589761678 1j)
>>> zeta(0.5+100000000j, derivative=4)
(-11967.3557309504640213060 2 - 218945.7817789262839266148j)
>>> zeta(1+10000000j)    # off the line
(2.8598464833325303370088 2 + 0.4918080474809818089039 86j)
>>> zeta(1+10000000j, derivative=1)
(-4.3338354946797915673205 - 0.0840533796260293363609 6103j)
>>> zeta(1+10000000j, derivative=4)
(453.2764822702057701894278 - 581.9636258327681891409 95j)
```

For investigation of the zeta function zeros, the Riemann-Siegel Z-function is often more convenient than working with the Riemann zeta function directly (see `siegelz()` (page 844)).

Some values of the Hurwitz zeta function:

```
>>> zeta(2, 3); -5./4 + pi**2/6
0.3949340668482264364724152
0.3949340668482264364724152
>>> zeta(2, (3,4)); pi**2 - 8*catalan
2.5418796476716064983976 63
2.5418796476716064983976 63
```

For positive integer values of s, the Hurwitz zeta function is equivalent to a polygamma function (except for a normalizing factor):

```
>>> zeta(4, (1,5)); psi(3, '1/5')/6
625.5408324774542966919938
625.5408324774542966919938
```

Evaluation of derivatives:

```
>>> zeta(0, 3+4j, 1); loggamma(3+4j) - ln(2*pi)/2
(-2.675565317808456852310934 + 4.742664438034657928194889j)
(-2.675565317808456852310934 + 4.742664438034657928194889j)
>>> zeta(2, 1, 20)
2432902008176640000.000242
>>> zeta(3+4j, 5.5+2j, 4)
(-0.140075548947797130681075 - 0.310926336027541325131363 4j)
>>> zeta(0.5+100000j, 1, 4)
(-10407.16081931495861539236 + 13777.78669862804508537384j)
>>> zeta(-100+0.5j, (1,3), derivative=4)
(4.00718082109982394270224 9e+79 + 4.91611795709259386832177 8e+78j)
```

Generating a Taylor series at $s = 2$ using derivatives:

```
>>> for k in range(11): print("%s * (s-2)^%i" % (zeta(2,1,k)/fac(k), k))
...
1.64493406684822643647241 5 * (s-2)^0
-0.937548254315843753702574 1 * (s-2)^1
0.994640117149450511710429 3 * (s-2)^2
-1.00002430047384081094065 7 * (s-2)^3
1.00006193307235256545751 2 * (s-2)^4
-1.00000686944393180640894 1 * (s-2)^5
1.00000017323376953182059 2 * (s-2)^6
-0.999999956998986849343239 9 * (s-2)^7
0.999999993721884450868420 6 * (s-2)^8
-0.999999999635501391660828 4 * (s-2)^9
1.00000000000461064502074 7 * (s-2)^10
```

Evaluation at zero and for negative integer s:

```
>>> zeta(0, 10)
-9.5
>>> zeta(-2, (2,3)); mpf(1)/81
0.0123456790123456790123456 8
0.0123456790123456790123456 8
>>> zeta(-3+4j, (5,4))
(0.289923603768269518208598 8 + 0.0656120616609175797311278 3j)
>>> zeta(-3.25, 1/pi)
-0.00051172696275744304943968 7
>>> zeta(-3.5, pi, 1)
11.15636039044000329470 9
>>> zeta(-100.5, (8,3))
-4.68162300487989766727122e+77
>>> zeta(-10.5, (-8,3))
(-0.0152191370444624660923797 9 + 29907.7251087424816160821 6j)
>>> zeta(-1000.5, (-8,3))
(1.03191194906233453820256 7e+1770 + 1.51955575055679421880472 4e+426j)
>>> zeta(-1+j, 3+4j)
(-16.3298835563080251088863 1 - 22.1770646580137403326138 3j)
>>> zeta(-1+j, 3+4j, 2)
(32.4898527639205664159405 5 - 51.1160446615739726704365 5j)
>>> diff(lambda s: zeta(s, 3+4j), -1+j, 2)
(32.4898527639205664159405 5 - 51.1160446615739726704365 5j)
```

References

1. http://mathworld.wolfram.com/RiemannZetaFunction.html
2. http://mathworld.wolfram.com/HurwitzZetaFunction.html
3. http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P155.pdf

Dirichlet L-series

altzeta()

mpmath.**altzeta**(s)

Gives the Dirichlet eta function, $\eta(s)$, also known as the alternating zeta function. This function is defined in analogy with the Riemann zeta function as providing the sum of

the alternating series

$$\eta(s) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k^s} = 1 - \frac{1}{2^s} + \frac{1}{3^s} - \frac{1}{4^s} + \dots$$

The eta function, unlike the Riemann zeta function, is an entire function, having a finite value for all complex s. The special case $\eta(1) = \log(2)$ gives the value of the alternating harmonic series.

The alternating zeta function may expressed using the Riemann zeta function as $\eta(s) = (1 - 2^{1-s})\zeta(s)$. It can also be expressed in terms of the Hurwitz zeta function, for example using `dirichlet()` (page 841) (see documentation for that function).

Examples

Some special values are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> altzeta(1)
0.693147180559945
>>> altzeta(0)
0.5
>>> altzeta(-1)
0.25
>>> altzeta(-2)
0.0
```

An example of a sum that can be computed more accurately and efficiently via `altzeta()` (page 839) than via numerical summation:

```
>>> sum(-(-1)**n / n**2.5 for n in range(1, 100))
0.86720495150398402
>>> altzeta(2.5)
0.867199889012184
```

At positive even integers, the Dirichlet eta function evaluates to a rational multiple of a power of π:

```
>>> altzeta(2)
0.822467033424113
>>> pi**2/12
0.822467033424113
```

Like the Riemann zeta function, $\eta(s)$, approaches 1 as s approaches positive infinity, although it does so from below rather than from above:

```
>>> altzeta(30)
0.999999999068682
>>> altzeta(inf)
1.0
>>> mp.pretty = False
>>> altzeta(1000, rounding='d')
mpf('0.99999999999999989')
>>> altzeta(1000, rounding='u')
mpf('1.0')
```

References

1. http://mathworld.wolfram.com/DirichletEtaFunction.html
2. http://en.wikipedia.org/wiki/Dirichlet_eta_function

dirichlet()

mpmath.**dirichlet**(*s*, *chi*, *derivative=0*)

Evaluates the Dirichlet L-function

$$L(s,\chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s}.$$

where χ is a periodic sequence of length q which should be supplied in the form of a list $[\chi(0), \chi(1), \ldots, \chi(q-1)]$. Strictly, χ should be a Dirichlet character, but any periodic sequence will work.

For example, `dirichlet(s, [1])` gives the ordinary Riemann zeta function and `dirichlet(s, [-1,1])` gives the alternating zeta function (Dirichlet eta function).

Also the derivative with respect to s (currently only a first derivative) can be evaluated.

Examples

The ordinary Riemann zeta function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> dirichlet(3, [1]); zeta(3)
1.202056903159594285399738
1.202056903159594285399738
>>> dirichlet(1, [1])
+inf
```

The alternating zeta function:

```
>>> dirichlet(1, [-1,1]); ln(2)
0.6931471805599453094172321
0.6931471805599453094172321
```

The following defines the Dirichlet beta function $\beta(s) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^s}$ and verifies several values of this function:

```
>>> B = lambda s, d=0: dirichlet(s, [0, 1, 0, -1], d)
>>> B(0); 1./2
0.5
0.5
>>> B(1); pi/4
0.7853981633974483096156609
0.7853981633974483096156609
>>> B(2); +catalan
0.9159655941772190150546035
0.9159655941772190150546035
>>> B(2,1); diff(B, 2)
0.08158073611659279510291217
0.08158073611659279510291217
>>> B(-1,1); 2*catalan/pi
0.5831218080616375602767689
0.5831218080616375602767689
>>> B(0,1); log(gamma(0.25)**2/(2*pi*sqrt(2)))
0.3915943927068367764719453
0.3915943927068367764719454
>>> B(1,1); 0.25*pi*(euler+2*ln2+3*ln(pi)-4*ln(gamma(0.25)))
0.1929013167969124293631898
0.1929013167969124293631898
```

A custom L-series of period 3:

```
>>> dirichlet(2, [2,0,1])
0.705971504783907809214683 1
>>> 2*nsum(lambda k: (3*k)**-2, [1,inf]) + \
...   nsum(lambda k: (3*k+2)**-2, [0,inf])
0.705971504783907809214683 1
```

Stieltjes constants

stieltjes()

mpmath.**stieltjes**(*n, a=1*)

For a nonnegative integer n, `stieltjes(n)` computes the n-th Stieltjes constant γ_n, defined as the n-th coefficient in the Laurent series expansion of the Riemann zeta function around the pole at $s = 1$. That is, we have:

$$\zeta(s) = \frac{1}{s-1} \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n (s-1)^n$$

More generally, `stieltjes(n, a)` gives the corresponding coefficient $\gamma_n(a)$ for the Hurwitz zeta function $\zeta(s, a)$ (with $\gamma_n = \gamma_n(1)$).

Examples

The zeroth Stieltjes constant is just Euler's constant γ:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> stieltjes(0)
0.577215664901533
```

Some more values are:

```
>>> stieltjes(1)
-0.0728158454836767
>>> stieltjes(10)
0.000205332814909065
>>> stieltjes(30)
0.00355772885557316
>>> stieltjes(1000)
-1.57095384420474e+486
>>> stieltjes(2000)
2.680424678918e+1109
>>> stieltjes(1, 2.5)
-0.23747539175716
```

An alternative way to compute γ_1:

```
>>> diff(extradps(15)(lambda x: 1/(x-1) - zeta(x)), 1)
-0.0728158454836767
```

`stieltjes()` (page 842) supports arbitrary precision evaluation:

```
>>> mp.dps = 50
>>> stieltjes(2)
-0.0096903631928723184845303860352125293590658061013408
```

Algorithm

stieltjes() (page 842) numerically evaluates the integral in the following representation due to Ainsworth, Howell and Coffey [1], [2]:

$$\gamma_n(a) = \frac{\log^n a}{2a} - \frac{\log^{n+1}(a)}{n+1} + \frac{2}{a}\Re\int_0^\infty \frac{(x/a-i)\log^n(a-ix)}{(1+x^2/a^2)(e^{2\pi x}-1)}dx.$$

For some reference values with $a = 1$, see e.g. [4].

References

1. O. R. Ainsworth & L. W. Howell, "An integral representation of the generalized Euler-Mascheroni constants", NASA Technical Paper 2456 (1985), http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19850014994_1985014994.pdf
2. M. W. Coffey, "The Stieltjes constants, their relation to the η_j coefficients, and representation of the Hurwitz zeta function", arXiv:0706.0343v1 http://arxiv.org/abs/0706.0343
3. http://mathworld.wolfram.com/StieltjesConstants.html
4. http://pi.lacim.uqam.ca/piDATA/stieltjesgamma.txt

Zeta function zeros These functions are used for the study of the Riemann zeta function in the critical strip.

zetazero()

mpmath.**zetazero**(*n*, *verbose=False*)

Computes the n-th nontrivial zero of $\zeta(s)$ on the critical line, i.e. returns an approximation of the n-th largest complex number $s = \frac{1}{2} + ti$ for which $\zeta(s) = 0$. Equivalently, the imaginary part t is a zero of the Z-function (siegelz() (page 844)).

Examples

The first few zeros:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> zetazero(1)
(0.5 + 14.13472514173469379045725j)
>>> zetazero(2)
(0.5 + 21.02203963877155499262848j)
>>> zetazero(20)
(0.5 + 77.14484006887480537268266j)
```

Verifying that the values are zeros:

```
>>> for n in range(1,5):
...     s = zetazero(n)
...     chop(zeta(s)), chop(siegelz(s.imag))
...
(0.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
```

Negative indices give the conjugate zeros ($n = 0$ is undefined):

```
>>> zetazero(-1)
(0.5 - 14.13472514173469379045725j)
```

zetazero() (page 843) supports arbitrarily large n and arbitrary precision:

```
>>> mp.dps = 15
>>> zetazero(1234567)
(0.5 + 727690.906948208j)
>>> mp.dps = 50
>>> zetazero(1234567)
(0.5 + 727690.9069482075392389420041147142092708393819935j)
>>> chop(zeta(_)/_)
0.0
```

with *info=True*, zetazero() (page 843) gives additional information:

```
>>> mp.dps = 15
>>> zetazero(542964976,info=True)
((0.5 + 209039046.578535j), [542964969, 542964978], 6, '(013111110)')
```

This means that the zero is between Gram points 542964969 and 542964978; it is the 6-th zero between them. Finally (01311110) is the pattern of zeros in this interval. The numbers indicate the number of zeros in each Gram interval (Rosser blocks between parenthesis). In this case there is only one Rosser block of length nine.

nzeros()

mpmath.**nzeros**(t)

Computes the number of zeros of the Riemann zeta function in $(0,1) \times (0,t]$, usually denoted by $N(t)$.

Examples

The first zero has imaginary part between 14 and 15:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nzeros(14)
0
>>> nzeros(15)
1
>>> zetazero(1)
(0.5 + 14.1347251417347j)
```

Some closely spaced zeros:

```
>>> nzeros(10**7)
21136125
>>> zetazero(21136125)
(0.5 + 9999999.32718175j)
>>> zetazero(21136126)
(0.5 + 10000000.2400236j)
>>> nzeros(545439823.215)
1500000001
>>> zetazero(1500000001)
(0.5 + 545439823.201985j)
>>> zetazero(1500000002)
(0.5 + 545439823.325697j)
```

This confirms the data given by J. van de Lune, H. J. J. te Riele and D. T. Winter in 1986.

siegelz()

mpmath.**siegelz**(t)

Computes the Z-function, also known as the Riemann-Siegel Z function,

$$Z(t) = e^{i\theta(t)}\zeta(1/2 + it)$$

where $\zeta(s)$ is the Riemann zeta function (zeta() (page 836)) and where $\theta(t)$ denotes the Riemann-Siegel theta function (see siegeltheta() (page 846)).

Evaluation is supported for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> siegelz(1)
-0.7363054628673177346778998
>>> siegelz(3+4j)
(-0.1852895764366314976003936 - 0.2773099198055652246992479j)
```

The first four derivatives are supported, using the optional *derivative* keyword argument:

```
>>> siegelz(1234567, derivative=3)
56.89689348495089294249178
>>> diff(siegelz, 1234567, n=3)
56.89689348495089294249178
```

The Z-function has a Maclaurin expansion:

```
>>> nprint(chop(taylor(siegelz, 0, 4)))
[-1.46035, 0.0, 2.73588, 0.0, -8.39357]
```

The Z-function $Z(t)$ is equal to $\pm|\zeta(s)|$ on the critical line $s = 1/2+it$ (i.e. for real arguments t to Z). Its zeros coincide with those of the Riemann zeta function:

```
>>> findroot(siegelz, 14)
14.13472514173469379045725
>>> findroot(siegelz, 20)
21.02203963877155499262848
>>> findroot(zeta, 0.5+14j)
(0.5 + 14.13472514173469379045725j)
>>> findroot(zeta, 0.5+20j)
(0.5 + 21.02203963877155499262848j)
```

Since the Z-function is real-valued on the critical line (and unlike $|\zeta(s)|$ analytic), it is useful for investigating the zeros of the Riemann zeta function. For example, one can use a root-finding algorithm based on sign changes:

```
>>> findroot(siegelz, [100, 200], solver='bisect')
176.4414342977104188888926
```

To locate roots, Gram points g_n which can be computed by grampoint() (page 846) are useful. If $(-1)^n Z(g_n)$ is positive for two consecutive n, then $Z(t)$ must have a zero between those points:

```
>>> g10 = grampoint(10)
>>> g11 = grampoint(11)
>>> (-1)**10 * siegelz(g10) > 0
True
>>> (-1)**11 * siegelz(g11) > 0
True
>>> findroot(siegelz, [g10, g11], solver='bisect')
56.44624769706339480436776
```

```
>>> g10, g11
(54.67523744685325626632663, 57.54516517954725443703014)
```

siegeltheta()

mpmath.**siegeltheta**(t)

Computes the Riemann-Siegel theta function,

$$\theta(t) = \frac{\log\Gamma\left(\frac{1+2it}{4}\right) - \log\Gamma\left(\frac{1-2it}{4}\right)}{2i} - \frac{\log\pi}{2}t.$$

The Riemann-Siegel theta function is important in providing the phase factor for the Z-function (see siegelz() (page 844)). Evaluation is supported for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> siegeltheta(0)
0.0
>>> siegeltheta(inf)
+inf
>>> siegeltheta(-inf)
-inf
>>> siegeltheta(1)
-1.767547952812290388302216
>>> siegeltheta(10+0.25j)
(-3.068638039426838572528867 + 0.05804937947429712998395177j)
```

Arbitrary derivatives may be computed with derivative = k

```
>>> siegeltheta(1234, derivative=2)
0.000405186407911405310947374 1
>>> diff(siegeltheta, 1234, n=2)
0.000405186407911405310947374 1
```

The Riemann-Siegel theta function has odd symmetry around $t = 0$, two local extreme points and three real roots including 0 (located symmetrically):

```
>>> nprint(chop(taylor(siegeltheta, 0, 5)))
[0.0, -2.68609, 0.0, 2.69433, 0.0, -6.40218]
>>> findroot(diffun(siegeltheta), 7)
6.289835988836902779665 09
>>> findroot(siegeltheta, 20)
17.84559954041086081682634
```

For large t, there is a famous asymptotic formula for $\theta(t)$, to first order given by:

```
>>> t = mpf(10**6)
>>> siegeltheta(t)
5488816.353078403444882823
>>> -t*log(2*pi/t)/2-t/2
5488816.745777464310273645
```

grampoint()

mpmath.**grampoint**(n)

Gives the n-th Gram point g_n, defined as the solution to the equation $\theta(g_n) = \pi n$ where $\theta(t)$ is the Riemann-Siegel theta function (siegeltheta() (page 846)).

The first few Gram points are:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> grampoint(0)
17.845599540410860816826 34
>>> grampoint(1)
23.170282701246309278996 64
>>> grampoint(2)
27.670182217816337960938 49
>>> grampoint(3)
31.717979954764053179551 49
```

Checking the definition:

```
>>> siegeltheta(grampoint(3))
9.424777960769379715387 93
>>> 3*pi
9.424777960769379715387 93
```

A large Gram point:

```
>>> grampoint(10**10)
3293531632.728335454561153
```

Gram points are useful when studying the Z-function (`siegelz()` (page 844)). See the documentation of that function for additional examples.

`grampoint()` (page 846) can solve the defining equation for nonintegral n. There is a fixed point where $g(x) = x$:

```
>>> findroot(lambda x: grampoint(x) - x, 10000)
9146.698193171459265866198
```

References

1. http://mathworld.wolfram.com/GramPoint.html

backlunds()

mpmath.**backlunds**(t)

Computes the function $S(t) = \arg \zeta(\frac{1}{2} + it)/\pi$.

See Titchmarsh Section 9.3 for details of the definition.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> backlunds(217.3)
0.16302205431184
```

Generally, the value is a small number. At Gram points it is an integer, frequently equal to 0:

```
>>> chop(backlunds(grampoint(200)))
0.0
>>> backlunds(extraprec(10)(grampoint)(211))
1.0
>>> backlunds(extraprec(10)(grampoint)(232))
-1.0
```

The number of zeros of the Riemann zeta function up to height t satisfies $N(t) = \theta(t)/\pi + 1 + S(t)$ (see :func:nzeros' and `siegeltheta()`):

```
>>> t = 1234.55
>>> nzeros(t)
842
>>> siegeltheta(t)/pi+1+backlunds(t)
842.0
```

Lerch transcendent

lerchphi()

mpmath.**lerchphi**(*z, s, a*)

Gives the Lerch transcendent, defined for $|z| < 1$ and $\Re a > 0$ by

$$\Phi(z, s, a) = \sum_{k=0}^{\infty} \frac{z^k}{(a+k)^s}$$

and generally by the recurrence $\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}$ along with the integral representation valid for $\Re a > 0$

$$\Phi(z, s, a) = \frac{1}{2a^s} + \int_0^{\infty} \frac{z^t}{(a+t)^s} dt - 2\int_0^{\infty} \frac{\sin(t \log z - s \arctan(t/a)}{(a^2+t^2)^{s/2}(e^{2\pi t}-1)} dt.$$

The Lerch transcendent generalizes the Hurwitz zeta function `zeta()` ($z = 1$) and the polylogarithm `polylog()` ($a = 1$).

Examples

Several evaluations in terms of simpler functions:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> lerchphi(-1,2,0.5); 4*catalan
3.663862376708876060218414
3.663862376708876060218414
>>> diff(lerchphi, (-1,-2,1), (0,1,0)); 7*zeta(3)/(4*pi**2)
0.2131391994087528954617607
0.2131391994087528954617607
>>> lerchphi(-4,1,1); log(5)/4
0.4023594781085250936501898
0.4023594781085250936501898
>>> lerchphi(-3+2j,1,0.5); 2*atanh(sqrt(-3+2j))/sqrt(-3+2j)
(1.142423447120257137774002 + 0.2118232380980201350495795j)
(1.142423447120257137774002 + 0.2118232380980201350495795j)
```

Evaluation works for complex arguments and $|z| \geq 1$:

```
>>> lerchphi(1+2j, 3-j, 4+2j)
(0.002025009957009908600539469 + 0.003327897536813558807438089j)
>>> lerchphi(-2,2,-2.5)
-12.28676272353094275265944
>>> lerchphi(10,10,10)
(-4.462130727102185701817349e-11 + 1.575172198981096218823481e-12j)
>>> lerchphi(10,10,-10.5)
(112658784011940.5605789002 + 498113185.5756221777743631j)
```

Some degenerate cases:

```
>>> lerchphi(0,1,2)
0.5
>>> lerchphi(0,1,-2)
-0.5
```

References

1. [DLMF] (page 1625) section 25.14

Polylogarithms and Clausen functions

polylog()

mpmath.**polylog**(s, z)

Computes the polylogarithm, defined by the sum

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}.$$

This series is convergent only for $|z| < 1$, so elsewhere the analytic continuation is implied.

The polylogarithm should not be confused with the logarithmic integral (also denoted by Li or li), which is implemented as `li()` (page 705).

Examples

The polylogarithm satisfies a huge number of functional identities. A sample of polylogarithm evaluations is shown below:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> polylog(1,0.5), log(2)
(0.693147180559945, 0.693147180559945)
>>> polylog(2,0.5), (pi**2-6*log(2)**2)/12
(0.582240526465012, 0.582240526465012)
>>> polylog(2,-phi), -log(phi)**2-pi**2/10
(-1.21852526068613, -1.21852526068613)
>>> polylog(3,0.5), 7*zeta(3)/8-pi**2*log(2)/12+log(2)**3/6
(0.53721319360804, 0.53721319360804)
```

`polylog()` (page 849) can evaluate the analytic continuation of the polylogarithm when s is an integer:

```
>>> polylog(2, 10)
(0.536301287357863 - 7.23378441241546j)
>>> polylog(2, -10)
-4.1982778868581
>>> polylog(2, 10j)
(-3.05968879432873 + 3.71678149306807j)
>>> polylog(-2, 10)
-0.150891632373114
>>> polylog(-2, -10)
0.067618332081142
>>> polylog(-2, 10j)
(0.0384353698579347 + 0.0912451798066779j)
```

Some more examples, with arguments on the unit circle (note that the series definition cannot be used for computation here):

```
>>> polylog(2,j)
(-0.205616758356028 + 0.915965594177219j)
>>> j*catalan-pi**2/48
(-0.205616758356028 + 0.915965594177219j)
>>> polylog(3,exp(2*pi*j/3))
(-0.534247512515375 + 0.765587078525922j)
>>> -4*zeta(3)/9 + 2*j*pi**3/81
(-0.534247512515375 + 0.765587078525921j)
```

Polylogarithms of different order are related by integration and differentiation:

```
>>> s, z = 3, 0.5
>>> polylog(s+1, z)
0.517479061673899
>>> quad(lambda t: polylog(s,t)/t, [0, z])
0.517479061673899
>>> z*diff(lambda t: polylog(s+2,t), z)
0.517479061673899
```

Taylor series expansions around $z = 0$ are:

```
>>> for n in range(-3, 4):
...     nprint(taylor(lambda x: polylog(n,x), 0, 5))
...
[0.0, 1.0, 8.0, 27.0, 64.0, 125.0]
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[0.0, 1.0, 0.5, 0.333333, 0.25, 0.2]
[0.0, 1.0, 0.25, 0.111111, 0.0625, 0.04]
[0.0, 1.0, 0.125, 0.037037, 0.015625, 0.008]
```

The series defining the polylogarithm is simultaneously a Taylor series and an L-series. For certain values of z, the polylogarithm reduces to a pure zeta function:

```
>>> polylog(pi, 1), zeta(pi)
(1.17624173838258, 1.17624173838258)
>>> polylog(pi, -1), -altzeta(pi)
(-0.909670702980385, -0.909670702980385)
```

Evaluation for arbitrary, nonintegral s is supported for z within the unit circle:

```
>>> polylog(3+4j, 0.25)
(0.24258605789446 - 0.00222938275488344j)
>>> nsum(lambda k: 0.25**k / k**(3+4j), [1,inf])
(0.24258605789446 - 0.00222938275488344j)
```

It is also currently supported outside of the unit circle for z not too large in magnitude:

```
>>> polylog(1+j, 20+40j)
(-7.1421172179728 - 3.92726697721369j)
>>> polylog(1+j, 200+400j)
Traceback (most recent call last):
  ...
NotImplementedError: polylog for arbitrary s and z
```

References

1. Richard Crandall, "Note on fast polylogarithm computation" http://people.reed.edu/~crandall/papers/Polylog.pdf
2. http://en.wikipedia.org/wiki/Polylogarithm
3. http://mathworld.wolfram.com/Polylogarithm.html

clsin()

mpmath.**clsin**(*s*, *z*)

Computes the Clausen sine function, defined formally by the series

$$\mathrm{Cl}_s(z) = \sum_{k=1}^{\infty} \frac{\sin(kz)}{k^s}.$$

The special case $\mathrm{Cl}_2(z)$ (i.e. `clsin(2,z)`) is the classical "Clausen function". More generally, the Clausen function is defined for complex s and z, even when the series does not converge. The Clausen function is related to the polylogarithm (`polylog()` (page 849)) as

$$\begin{aligned}\mathrm{Cl}_s(z) &= \frac{1}{2i}\left(\mathrm{Li}_s\left(e^{iz}\right) - \mathrm{Li}_s\left(e^{-iz}\right)\right) \\ &= \mathrm{Im}\left[\mathrm{Li}_s(e^{iz})\right] \quad (s, z \in \mathbb{R}),\end{aligned}$$

and this representation can be taken to provide the analytic continuation of the series. The complementary function `clcos()` (page 853) gives the corresponding cosine sum.

Examples

Evaluation for arbitrarily chosen s and z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> s, z = 3, 4
>>> clsin(s, z); nsum(lambda k: sin(z*k)/k**s, [1,inf])
-0.6533010136329338746275795
-0.6533010136329338746275795
```

Using $z + \pi$ instead of z gives an alternating series:

```
>>> clsin(s, z+pi)
0.8860032351260589402871624
>>> nsum(lambda k: (-1)**k*sin(z*k)/k**s, [1,inf])
0.8860032351260589402871624
```

With $s = 1$, the sum can be expressed in closed form using elementary functions:

```
>>> z = 1 + sqrt(3)
>>> clsin(1, z)
0.2047709230104579724675985
>>> chop((log(1-exp(-j*z)) - log(1-exp(j*z)))/(2*j))
0.2047709230104579724675985
>>> nsum(lambda k: sin(k*z)/k, [1,inf])
0.2047709230104579724675985
```

The classical Clausen function $\mathrm{Cl}_2(\theta)$ gives the value of the integral $\int_0^\theta -\ln(2\sin(x/2))dx$ for $0 < \theta < 2\pi$:

```
>>> cl2 = lambda t: clsin(2, t)
>>> cl2(3.5)
-0.2465045302347694216534255
```

```
>>> -quad(lambda x: ln(2*sin(0.5*x)), [0, 3.5])
-0.24650453023476942165342 55
```

This function is symmetric about $\theta = \pi$ with zeros and extreme points:

```
>>> cl2(0); cl2(pi/3); chop(cl2(pi)); cl2(5*pi/3); chop(cl2(2*pi))
0.0
1.01494160640965362502120 3
0.0
-1.01494160640965362502120 3
0.0
```

Catalan's constant is a special value:

```
>>> cl2(pi/2)
0.915965594177219015054603 5
>>> +catalan
0.915965594177219015054603 5
```

The Clausen sine function can be expressed in closed form when s is an odd integer (becoming zero when $s < 0$):

```
>>> z = 1 + sqrt(2)
>>> clsin(1, z); (pi-z)/2
0.36368954560834909483047 73
0.36368954560834909483047 73
>>> clsin(3, z); pi**2/6*z - pi*z**2/4 + z**3/12
0.56617515844511449917071 61
0.56617515844511449917071 61
>>> clsin(-1, z)
0.0
>>> clsin(-3, z)
0.0
```

It can also be expressed in closed form for even integer $s \leq 0$, providing a finite sum for series such as $\sin(z) + \sin(2z) + \sin(3z) + \ldots$:

```
>>> z = 1 + sqrt(2)
>>> clsin(0, z)
0.19031050295075138812758 65
>>> cot(z/2)/2
0.19031050295075138812758 65
>>> clsin(-2, z)
-0.10894061638415488175813 92
>>> -cot(z/2)*csc(z/2)**2/4
-0.10894061638415488175813 92
```

Call with `pi=True` to multiply z by π exactly:

```
>>> clsin(3, 3*pi)
-8.8923162249680724247328 98e-26
>>> clsin(3, 3, pi=True)
0.0
```

Evaluation for complex s, z in a nonconvergent case:

```
>>> s, z = -1-j, 1+2j
>>> clsin(s, z)
(-0.59307948011737900251603 4 + 0.90386442333678682733624 46j)
>>> extraprec(20)(nsum)(lambda k: sin(k*z)/k**s, [1,inf])
(-0.59307948011737900251603 4 + 0.90386442333678682733624 46j)
```

clcos()

mpmath.**clcos**(s, z)

Computes the Clausen cosine function, defined formally by the series

$$\widetilde{\mathrm{Cl}}_s(z) = \sum_{k=1}^{\infty} \frac{\cos(kz)}{k^s}.$$

This function is complementary to the Clausen sine function clsin() (page 851). In terms of the polylogarithm,

$$\begin{aligned}\widetilde{\mathrm{Cl}}_s(z) &= \frac{1}{2}\left(\mathrm{Li}_s\left(e^{iz}\right) + \mathrm{Li}_s\left(e^{-iz}\right)\right) \\ &= \mathrm{Re}\left[\mathrm{Li}_s(e^{iz})\right] \quad (s, z \in \mathbb{R}).\end{aligned}$$

Examples

Evaluation for arbitrarily chosen s and z:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> s, z = 3, 4
>>> clcos(s, z); nsum(lambda k: cos(z*k)/k**s, [1,inf])
-0.6518926267198991308332759
-0.6518926267198991308332759
```

Using $z + \pi$ instead of z gives an alternating series:

```
>>> s, z = 3, 0.5
>>> clcos(s, z+pi)
-0.8155530586502260817855618
>>> nsum(lambda k: (-1)**k*cos(z*k)/k**s, [1,inf])
-0.8155530586502260817855618
```

With $s = 1$, the sum can be expressed in closed form using elementary functions:

```
>>> z = 1 + sqrt(3)
>>> clcos(1, z)
-0.6720334373369714849797918
>>> chop(-0.5*(log(1-exp(j*z))+log(1-exp(-j*z))))
-0.6720334373369714849797918
>>> -log(abs(2*sin(0.5*z)))    # Equivalent to above when z is real
-0.6720334373369714849797918
>>> nsum(lambda k: cos(k*z)/k, [1,inf])
-0.6720334373369714849797918
```

It can also be expressed in closed form when s is an even integer. For example,

```
>>> clcos(2,z)
-0.7805359025135583118863007
>>> pi**2/6 - pi*z/2 + z**2/4
-0.7805359025135583118863007
```

The case $s = 0$ gives the renormalized sum of $\cos(z)+\cos(2z)+\cos(3z)+\ldots$ (which happens to be the same for any value of z):

```
>>> clcos(0, z)
-0.5
>>> nsum(lambda k: cos(k*z), [1,inf])
-0.5
```

Also the sums

$$\cos(z) + 2\cos(2z) + 3\cos(3z) + \ldots$$

and

$$\cos(z) + 2^n\cos(2z) + 3^n\cos(3z) + \ldots$$

for higher integer powers $n = -s$ can be done in closed form. They are zero when n is positive and even (s negative and even):

```
>>> clcos(-1, z); 1/(2*cos(z)-2)
-0.260782937524054248069412 6
-0.260782937524054248069412 6
>>> clcos(-3, z); (2+cos(z))*csc(z/2)**4/8
0.147263505497994439084800 6
0.147263505497994439084800 6
>>> clcos(-2, z); clcos(-4, z); clcos(-6, z)
0.0
0.0
0.0
```

With $z = \pi$, the series reduces to that of the Riemann zeta function (more generally, if $z = p\pi/q$, it is a finite sum over Hurwitz zeta function values):

```
>>> clcos(2.5, 0); zeta(2.5)
1.341487257250917179756 77
1.341487257250917179756 77
>>> clcos(2.5, pi); -altzeta(2.5)
-0.867199889012184138191 3472
-0.867199889012184138191 3472
```

Call with `pi=True` to multiply z by π exactly:

```
>>> clcos(-3, 2*pi)
2.997921055881167659267063e+102
>>> clcos(-3, 2, pi=True)
0.008333333333333333333333333
```

Evaluation for complex s, z in a nonconvergent case:

```
>>> s, z = -1-j, 1+2j
>>> clcos(s, z)
(0.940743012156225147613 6807 + 0.715826296033590204557054j)
>>> extraprec(20)(nsum)(lambda k: cos(k*z)/k**s, [1,inf])
(0.940743012156225147613 6807 + 0.715826296033590204557054j)
```

polyexp()

mpmath.**polyexp**(s, z)

Evaluates the polyexponential function, defined for arbitrary complex s, z by the series

$$E_s(z) = \sum_{k=1}^{\infty} \frac{k^s}{k!} z^k.$$

$E_s(z)$ is constructed from the exponential function analogously to how the polylogarithm is constructed from the ordinary logarithm; as a function of s (with z fixed), E_s is an L-series It is an entire function of both s and z.

The polyexponential function provides a generalization of the Bell polynomials $B_n(x)$ (see `bell()` (page 865)) to noninteger orders n. In terms of the Bell polynomials,

$$E_s(z) = e^z B_s(z) - \operatorname{sinc}(\pi s).$$

Note that $B_n(x)$ and $e^{-x} E_n(x)$ are identical if n is a nonzero integer, but not otherwise. In particular, they differ at $n = 0$.

Examples

Evaluating a series:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> nsum(lambda k: sqrt(k)/fac(k), [1,inf])
2.101755547733791780315904
>>> polyexp(0.5,1)
2.101755547733791780315904
```

Evaluation for arbitrary arguments:

```
>>> polyexp(-3-4j, 2.5+2j)
(2.351660261190434618268706 + 1.202966666673054671364215j)
```

Evaluation is accurate for tiny function values:

```
>>> polyexp(4, -100)
3.499471750566824369520223e-36
```

If n is a nonpositive integer, E_n reduces to a special instance of the hypergeometric function ${}_pF_q$:

```
>>> n = 3
>>> x = pi
>>> polyexp(-n,x)
4.042192318847986561771779
>>> x*hyper([1]*(n+1), [2]*(n+1), x)
4.042192318847986561771779
```

Zeta function variants

primezeta()

mpmath.**primezeta**(s)

Computes the prime zeta function, which is defined in analogy with the Riemann zeta function (`zeta()` (page 836)) as

$$P(s) = \sum_p \frac{1}{p^s}$$

where the sum is taken over all prime numbers p. Although this sum only converges for $\operatorname{Re}(s) > 1$, the function is defined by analytic continuation in the half-plane $\operatorname{Re}(s) > 0$.

Examples

Arbitrary-precision evaluation for real and complex arguments is supported:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> primezeta(2)
0.452247420041065498506543364832
>>> primezeta(pi)
0.154837526988402842720364973 97
>>> mp.dps = 50
>>> primezeta(3)
0.17476263929944353642311331466570670097541212192615
>>> mp.dps = 20
>>> primezeta(3+4j)
(-0.12085382601645763295 - 0.013370403397787023602j)
```

The prime zeta function has a logarithmic pole at $s = 1$, with residue equal to the difference of the Mertens and Euler constants:

```
>>> primezeta(1)
+inf
>>> extradps(25)(lambda x: primezeta(1+x)+log(x))(+eps)
-0.31571845205389007685
>>> mertens-euler
-0.31571845205389007685
```

The analytic continuation to $0 < \mathrm{Re}(s) \leq 1$ is implemented. In this strip the function exhibits very complex behavior; on the unit interval, it has poles at $1/n$ for every squarefree integer n:

```
>>> primezeta(0.5)         # Pole at s = 1/2
(-inf + 3.141592653589793238 5j)
>>> primezeta(0.25)
(-1.041610680175726903 6 + 0.52359877559829887308j)
>>> primezeta(0.5+10j)
(0.54892423556409790529 + 0.45626803423487934264j)
```

Although evaluation works in principle for any $\mathrm{Re}(s) > 0$, it should be noted that the evaluation time increases exponentially as s approaches the imaginary axis.

For large $\mathrm{Re}(s)$, $P(s)$ is asymptotic to 2^{-s}:

```
>>> primezeta(inf)
0.0
>>> primezeta(10), mpf(2)**-10
(0.0009936035744369802178 6, 0.0009765625)
>>> primezeta(1000)
9.3326361850321887899e-302
>>> primezeta(1000+1000j)
(-3.8565440833654995949e-302 - 8.4985390447553234305e-302j)
```

References

Carl-Erik Froberg, "On the prime zeta function", BIT 8 (1968), pp. 187-202.

secondzeta()

mpmath.**secondzeta**(*s*, *a=0.015*, ***kwargs*)

Evaluates the secondary zeta function $Z(s)$, defined for $\mathrm{Re}(s) > 1$ by

$$Z(s) = \sum_{n=1}^{\infty} \frac{1}{\tau_n^s}$$

where $\frac{1}{2} + i\tau_n$ runs through the zeros of $\zeta(s)$ with imaginary part positive.

$Z(s)$ extends to a meromorphic function on $\mathbb{C}$ with a double pole at $s = 1$ and simple poles at the points $-2n$ for $n = 0$, 1, 2, ...

Examples

```
>>> from mpmath import *
>>> mp.pretty = True; mp.dps = 15
>>> secondzeta(2)
0.023104993115419
>>> xi = lambda s: 0.5*s*(s-1)*pi**(-0.5*s)*gamma(0.5*s)*zeta(s)
>>> Xi = lambda t: xi(0.5+t*j)
>>> -0.5*diff(Xi,0,n=2)/Xi(0)
(0.023104993115419 + 0.0j)
```

We may ask for an approximate error value:

```
>>> secondzeta(0.5+100j, error=True)
((-0.216272011276718 - 0.844952708937228j), 2.22044604925031e-16)
```

The function has poles at the negative odd integers, and dyadic rational values at the negative even integers:

```
>>> mp.dps = 30
>>> secondzeta(-8)
-0.67236328125
>>> secondzeta(-7)
+inf
```

Implementation notes

The function is computed as sum of four terms $Z(s) = A(s) - P(s) + E(s) - S(s)$ respectively main, prime, exponential and singular terms. The main term $A(s)$ is computed from the zeros of zeta. The prime term depends on the von Mangoldt function. The singular term is responsible for the poles of the function.

The four terms depends on a small parameter a. We may change the value of a. Theoretically this has no effect on the sum of the four terms, but in practice may be important.

A smaller value of the parameter a makes $A(s)$ depend on a smaller number of zeros of zeta, but $P(s)$ uses more values of von Mangoldt function.

We may also add a verbose option to obtain data about the values of the four terms.

```
>>> mp.dps = 10
>>> secondzeta(0.5 + 40j, error=True, verbose=True)
main term = (-30190318549.138656312556 - 13964804384.624622876523j)
    computed using 19 zeros of zeta
prime term = (132717176.89212754625045 + 188980555.17563978290601j)
    computed using 9 values of the von Mangoldt function
exponential term = (542447428666.07179812536 + 362434922978.80192435203j)
singular term = (512124392939.98154322355 + 348281138038.65531023921j)
((0.059471043 + 0.3463514534j), 1.455191523e-11)

>>> secondzeta(0.5 + 40j, a=0.04, error=True, verbose=True)
main term = (-151962888.19606243907725 - 217930683.90210294051982j)
    computed using 9 zeros of zeta
prime term = (2476659342.3038722372461 + 28711581821.921627163136j)
    computed using 37 values of the von Mangoldt function
exponential term = (178506047114.7838188264 + 819674143244.45677330576j)
```

```
singular term = (175877424884.22441310708 + 790744630738.28669174871j)
((0.059471043 + 0.3463514534j), 1.455191523e-11)
```

Notice the great cancellation between the four terms. Changing a, the four terms are very different numbers but the cancellation gives the good value of Z(s).

References

A. Voros, Zeta functions for the Riemann zeros, Ann. Institute Fourier, 53, (2003) 665–699.

A. Voros, Zeta functions over Zeros of Zeta Functions, Lecture Notes of the Unione Matematica Italiana, Springer, 2009.

Number-theoretical, combinatorial and integer functions

For factorial-type functions, including binomial coefficients, double factorials, etc., see the separate section *Factorials and gamma functions* (page 683).

Fibonacci numbers

fibonacci()/fib()

mpmath.**fibonacci**(*n*, ***kwargs*)

`fibonacci(n)` computes the n-th Fibonacci number, $F(n)$. The Fibonacci numbers are defined by the recurrence $F(n) = F(n-1)+F(n-2)$ with the initial values $F(0) = 0$, $F(1) = 1$. `fibonacci()` (page 858) extends this definition to arbitrary real and complex arguments using the formula

$$F(z) = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

where ϕ is the golden ratio. `fibonacci()` (page 858) also uses this continuous formula to compute $F(n)$ for extremely large n, where calculating the exact integer would be wasteful.

For convenience, `fib()` is available as an alias for `fibonacci()` (page 858).

Basic examples

Some small Fibonacci numbers are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for i in range(10):
...     print(fibonacci(i))
...
0.0
1.0
1.0
2.0
3.0
5.0
8.0
13.0
21.0
34.0
```

```
>>> fibonacci(50)
12586269025.0
```

The recurrence for $F(n)$ extends backwards to negative n:

```
>>> for i in range(10):
...     print(fibonacci(-i))
...
0.0
1.0
-1.0
2.0
-3.0
5.0
-8.0
13.0
-21.0
34.0
```

Large Fibonacci numbers will be computed approximately unless the precision is set high enough:

```
>>> fib(200)
2.8057117299251e+41
>>> mp.dps = 45
>>> fib(200)
280571172992510140037611932413038677189525.0
```

`fibonacci()` (page 858) can compute approximate Fibonacci numbers of stupendous size:

```
>>> mp.dps = 15
>>> fibonacci(10**25)
3.49052338550226e+208987640249978733769272 0
```

Real and complex arguments

The extended Fibonacci function is an analytic function. The property $F(z) = F(z-1) + F(z-2)$ holds for arbitrary z:

```
>>> mp.dps = 15
>>> fib(pi)
2.1170270579161
>>> fib(pi-1) + fib(pi-2)
2.1170270579161
>>> fib(3+4j)
(-5248.51130728372 - 14195.962288353j)
>>> fib(2+4j) + fib(1+4j)
(-5248.51130728372 - 14195.962288353j)
```

The Fibonacci function has infinitely many roots on the negative half-real axis. The first root is at 0, the second is close to -0.18, and then there are infinitely many roots that asymptotically approach $-n + 1/2$:

```
>>> findroot(fib, -0.2)
-0.183802359692956
>>> findroot(fib, -2)
-1.57077646820395
>>> findroot(fib, -17)
-16.4999999596115
```

```
>>> findroot(fib, -24)
-23.5000000000479
```

Mathematical relationships

For large n, $F(n+1)/F(n)$ approaches the golden ratio:

```
>>> mp.dps = 50
>>> fibonacci(101)/fibonacci(100)
1.6180339887498948482045868343656381177203127439638
>>> +phi
1.6180339887498948482045868343656381177203091798058
```

The sum of reciprocal Fibonacci numbers converges to an irrational number for which no closed form expression is known:

```
>>> mp.dps = 15
>>> nsum(lambda n: 1/fib(n), [1, inf])
3.35988566624318
```

Amazingly, however, the sum of odd-index reciprocal Fibonacci numbers can be expressed in terms of a Jacobi theta function:

```
>>> nsum(lambda n: 1/fib(2*n+1), [0, inf])
1.82451515740692
>>> sqrt(5)*jtheta(2,0,(3-sqrt(5))/2)**2/4
1.82451515740692
```

Some related sums can be done in closed form:

```
>>> nsum(lambda k: 1/(1+fib(2*k+1)), [0, inf])
1.11803398874989
>>> phi - 0.5
1.11803398874989
>>> f = lambda k:(-1)**(k+1) / sum(fib(n)**2 for n in range(1,int(k+1)))
>>> nsum(f, [1, inf])
0.618033988749895
>>> phi-1
0.618033988749895
```

References

1. http://mathworld.wolfram.com/FibonacciNumber.html

Bernoulli numbers and polynomials

bernoulli()

mpmath.**bernoulli**(n)

Computes the nth Bernoulli number, B_n, for any integer $n \geq 0$.

The Bernoulli numbers are rational numbers, but this function returns a floating-point approximation. To obtain an exact fraction, use `bernfrac()` (page 861) instead.

Examples

Numerical values of the first few Bernoulli numbers:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(15):
```

```
...     print("%s %s" % (n, bernoulli(n)))
...
0 1.0
1 -0.5
2 0.166666666666667
3 0.0
4 -0.0333333333333333
5 0.0
6 0.0238095238095238
7 0.0
8 -0.0333333333333333
9 0.0
10 0.0757575757575758
11 0.0
12 -0.253113553113553
13 0.0
14 1.16666666666667
```

Bernoulli numbers can be approximated with arbitrary precision:

```
>>> mp.dps = 50
>>> bernoulli(100)
-2.8382249570693706959264156336481764738284680928013e+78
```

Arbitrarily large n are supported:

```
>>> mp.dps = 15
>>> bernoulli(10**20 + 2)
3.09136296657021e+187675256497386331232 7
```

The Bernoulli numbers are related to the Riemann zeta function at integer arguments:

```
>>> -bernoulli(8) * (2*pi)**8 / (2*fac(8))
1.00407735619794
>>> zeta(8)
1.00407735619794
```

Algorithm

For small n ($n < 3000$) `bernoulli()` (page 860) uses a recurrence formula due to Ramanujan. All results in this range are cached, so sequential computation of small Bernoulli numbers is guaranteed to be fast.

For larger n, B_n is evaluated in terms of the Riemann zeta function.

bernfrac()

mpmath.**bernfrac**(*n*)

Returns a tuple of integers (p, q) such that $p/q = B_n$ exactly, where B_n denotes the n-th Bernoulli number. The fraction is always reduced to lowest terms. Note that for $n > 1$ and n odd, $B_n = 0$, and $(0, 1)$ is returned.

Examples

The first few Bernoulli numbers are exactly:

```
>>> from mpmath import *
>>> for n in range(15):
...     p, q = bernfrac(n)
...     print("%s %s/%s" % (n, p, q))
...
```

```
0 1/1
1 -1/2
2 1/6
3 0/1
4 -1/30
5 0/1
6 1/42
7 0/1
8 -1/30
9 0/1
10 5/66
11 0/1
12 -691/2730
13 0/1
14 7/6
```

This function works for arbitrarily large n:

```
>>> p, q = bernfrac(10**4)
>>> print(q)
2338224387510
>>> print(len(str(p)))
27692
>>> mp.dps = 15
>>> print(mpf(p) / q)
-9.04942396360948e+27677
>>> print(bernoulli(10**4))
-9.04942396360948e+27677
```

Note: bernoulli() (page 860) computes a floating-point approximation directly, without computing the exact fraction first. This is much faster for large n.

Algorithm

bernfrac() (page 861) works by computing the value of B_n numerically and then using the von Staudt-Clausen theorem [1] to reconstruct the exact fraction. For large n, this is significantly faster than computing $B_1, B_2, \ldots, B_2$ recursively with exact arithmetic. The implementation has been tested for $n = 10^m$ up to $m = 6$.

In practice, bernfrac() (page 861) appears to be about three times slower than the specialized program calcbn.exe [2]

References

1. MathWorld, von Staudt-Clausen Theorem: http://mathworld.wolfram.com/vonStaudt-ClausenTheorem.html
2. The Bernoulli Number Page: http://www.bernoulli.org/

bernpoly()

mpmath.**bernpoly**(n, z)

Evaluates the Bernoulli polynomial $B_n(z)$.

The first few Bernoulli polynomials are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(6):
```

```
...     nprint(chop(taylor(lambda x: bernpoly(n,x), 0, n)))
...
[1.0]
[-0.5, 1.0]
[0.166667, -1.0, 1.0]
[0.0, 0.5, -1.5, 1.0]
[-0.0333333, 0.0, 1.0, -2.0, 1.0]
[0.0, -0.166667, 0.0, 1.66667, -2.5, 1.0]
```

At $z = 0$, the Bernoulli polynomial evaluates to a Bernoulli number (see `bernoulli()` (page 860)):

```
>>> bernpoly(12, 0), bernoulli(12)
(-0.253113553113553, -0.253113553113553)
>>> bernpoly(13, 0), bernoulli(13)
(0.0, 0.0)
```

Evaluation is accurate for large n and small z:

```
>>> mp.dps = 25
>>> bernpoly(100, 0.5)
2.838224957069370695926416e+78
>>> bernpoly(1000, 10.5)
5.31870446941552203648291 4e+1769
```

Euler numbers and polynomials

eulernum()

mpmath.**eulernum**(n)

Gives the n-th Euler number, defined as the n-th derivative of $\operatorname{sech}(t) = 1/\cosh(t)$ evaluated at $t = 0$. Equivalently, the Euler numbers give the coefficients of the Taylor series

$$\operatorname{sech}(t) = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n.$$

The Euler numbers are closely related to Bernoulli numbers and Bernoulli polynomials. They can also be evaluated in terms of Euler polynomials (see `eulerpoly()` (page 864)) as $E_n = 2^n E_n(1/2)$.

Examples

Computing the first few Euler numbers and verifying that they agree with the Taylor series:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> [eulernum(n) for n in range(11)]
[1.0, 0.0, -1.0, 0.0, 5.0, 0.0, -61.0, 0.0, 1385.0, 0.0, -50521.0]
>>> chop(diffs(sech, 0, 10))
[1.0, 0.0, -1.0, 0.0, 5.0, 0.0, -61.0, 0.0, 1385.0, 0.0, -50521.0]
```

Euler numbers grow very rapidly. `eulernum()` (page 863) efficiently computes numerical approximations for large indices:

```
>>> eulernum(50)
-6.05328524818862189631438 4e+54
>>> eulernum(1000)
```

```
3.887561841253070615257336e+2371
>>> eulernum(10**20)
4.346791453661149089338186e+1936958564106659551331
```

Comparing with an asymptotic formula for the Euler numbers:

```
>>> n = 10**5
>>> (-1)**(n//2) * 8 * sqrt(n/(2*pi)) * (2*n/(pi*e))**n
3.699190630174323628056 63e+436961
>>> eulernum(n)
3.699193712834466537941283e+436961
```

Pass `exact=True` to obtain exact values of Euler numbers as integers:

```
>>> print(eulernum(50, exact=True))
-605328524818862189631438378511164908810349822514681 5121
>>> print(eulernum(200, exact=True) % 10**10)
1925859625
>>> eulernum(1001, exact=True)
0
```

eulerpoly()

mpmath.**eulerpoly**(*n*, *z*)

Evaluates the Euler polynomial $E_n(z)$, defined by the generating function representation

$$\frac{2e^{zt}}{e^t+1} = \sum_{n=0}^{\infty} E_n(z)\frac{t^n}{n!}.$$

The Euler polynomials may also be represented in terms of Bernoulli polynomials (see `bernpoly()` (page 862)) using various formulas, for example

$$E_n(z) = \frac{2}{n+1}\left(B_n(z) - 2^{n+1}B_n\left(\frac{z}{2}\right)\right).$$

Special values include the Euler numbers $E_n = 2^n E_n(1/2)$ (see `eulernum()` (page 863)).

Examples

Computing the coefficients of the first few Euler polynomials:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> for n in range(6):
...     chop(taylor(lambda z: eulerpoly(n,z), 0, n))
...
[1.0]
[-0.5, 1.0]
[0.0, -1.0, 1.0]
[0.25, 0.0, -1.5, 1.0]
[0.0, 1.0, 0.0, -2.0, 1.0]
[-0.5, 0.0, 2.5, 0.0, -2.5, 1.0]
```

Evaluation for arbitrary z:

```
>>> eulerpoly(2,3)
6.0
>>> eulerpoly(5,4)
423.5
>>> eulerpoly(35, 11111111112)
```

```
3.994957561486776072734601e+351
>>> eulerpoly(4, 10+20j)
(-47990.0 - 235980.0j)
>>> eulerpoly(2, '-3.5e-5')
0.000035001225
>>> eulerpoly(3, 0.5)
0.0
>>> eulerpoly(55, -10**80)
-1.0e+4400
>>> eulerpoly(5, -inf)
-inf
>>> eulerpoly(6, -inf)
+inf
```

Computing Euler numbers:

```
>>> 2**26 * eulerpoly(26,0.5)
-4087072509293123892361.0
>>> eulernum(26)
-4087072509293123892361.0
```

Evaluation is accurate for large n and small z:

```
>>> eulerpoly(100, 0.5)
2.290479999881941141779 43e+108
>>> eulerpoly(1000, 10.5)
3.628120031122876847764566e+2070
>>> eulerpoly(10000, 10.5)
1.149364285543783412210773e+30688
```

Bell numbers and polynomials

bell()

mpmath.**bell**(*n*, *x*)

For n a nonnegative integer, `bell(n,x)` evaluates the Bell polynomial $B_n(x)$, the first few of which are

$$B_0(x) = 1$$
$$B_1(x) = x$$
$$B_2(x) = x^2 + x$$
$$B_3(x) = x^3 + 3x^2 + x$$

If $x = 1$ or `bell()` (page 865) is called with only one argument, it gives the n-th Bell number B_n, which is the number of partitions of a set with n elements. By setting the precision to at least $\log_{10} B_n$ digits, `bell()` (page 865) provides fast calculation of exact Bell numbers.

In general, `bell()` (page 865) computes

$$B_n(x) = e^{-x} \left(\operatorname{sinc}(\pi n) + E_n(x)\right)$$

where $E_n(x)$ is the generalized exponential function implemented by `polyexp()` (page 854). This is an extension of Dobinski's formula [1], where the modification is the sinc term ensuring that $B_n(x)$ is continuous in n; `bell()` (page 865) can thus be evaluated, differentiated, etc for arbitrary complex arguments.

Examples

Simple evaluations:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> bell(0, 2.5)
1.0
>>> bell(1, 2.5)
2.5
>>> bell(2, 2.5)
8.75
```

Evaluation for arbitrary complex arguments:

```
>>> bell(5.75+1j, 2-3j)
(-10767.71345136587098445143 - 15449.55065599872579097221j)
```

The first few Bell polynomials:

```
>>> for k in range(7):
...     nprint(taylor(lambda x: bell(k,x), 0, k))
...
[1.0]
[0.0, 1.0]
[0.0, 1.0, 1.0]
[0.0, 1.0, 3.0, 1.0]
[0.0, 1.0, 7.0, 6.0, 1.0]
[0.0, 1.0, 15.0, 25.0, 10.0, 1.0]
[0.0, 1.0, 31.0, 90.0, 65.0, 15.0, 1.0]
```

The first few Bell numbers and complementary Bell numbers:

```
>>> [int(bell(k)) for k in range(10)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
>>> [int(bell(k,-1)) for k in range(10)]
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

Large Bell numbers:

```
>>> mp.dps = 50
>>> bell(50)
185724268771078270438257767181908917499221852770.0
>>> bell(50,-1)
-29113173035759403920216141265491160286912.0
```

Some even larger values:

```
>>> mp.dps = 25
>>> bell(1000,-1)
-1.237132026969293954162816e+1869
>>> bell(1000)
2.989901335682408421480422e+1927
>>> bell(1000,2)
6.591553486811969380442171e+1987
>>> bell(1000,100.5)
9.101014101401543575679639e+2529
```

A determinant identity satisfied by Bell numbers:

```
>>> mp.dps = 15
>>> N = 8
>>> det([[bell(k+j) for j in range(N)] for k in range(N)])
125411328000.0
>>> superfac(N-1)
125411328000.0
```

References

1. http://mathworld.wolfram.com/DobinskisFormula.html

Prime counting functions

primepi()

mpmath.**primepi**(x)

Evaluates the prime counting function, $\pi(x)$, which gives the number of primes less than or equal to x. The argument x may be fractional.

The prime counting function is very expensive to evaluate precisely for large x, and the present implementation is not optimized in any way. For numerical approximation of the prime counting function, it is better to use primepi2() (page 867) or riemannr() (page 868).

Some values of the prime counting function:

```
>>> from mpmath import *
>>> [primepi(k) for k in range(20)]
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 8]
>>> primepi(3.5)
2
>>> primepi(100000)
9592
```

primepi2()

mpmath.**primepi2**(x)

Returns an interval (as an mpi instance) providing bounds for the value of the prime counting function $\pi(x)$. For small x, primepi2() (page 867) returns an exact interval based on the output of primepi() (page 867). For $x > 2656$, a loose interval based on Schoenfeld's inequality

$$|\pi(x) - \mathrm{li}(x)| < \frac{\sqrt{x}\log x}{8\pi}$$

is returned. This estimate is rigorous assuming the truth of the Riemann hypothesis, and can be computed very quickly.

Examples

Exact values of the prime counting function for small x:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> iv.dps = 15; iv.pretty = True
>>> primepi2(10)
[4.0, 4.0]
>>> primepi2(100)
[25.0, 25.0]
```

```
>>> primepi2(1000)
[168.0, 168.0]
```

Loose intervals are generated for moderately large x:

```
>>> primepi2(10000), primepi(10000)
([1209.0, 1283.0], 1229)
>>> primepi2(50000), primepi(50000)
([5070.0, 5263.0], 5133)
```

As x increases, the absolute error gets worse while the relative error improves. The exact value of $\pi(10^{23})$ is 1925320391606803968923, and `primepi2()` (page 867) gives 9 significant digits:

```
>>> p = primepi2(10**23)
>>> p
[1.9253203909477020467e+21, 1.925320392280406229e+21]
>>> mpf(p.delta) / mpf(p.a)
6.9219865355293e-10
```

A more precise, nonrigorous estimate for $\pi(x)$ can be obtained using the Riemann R function (`riemannr()` (page 868)). For large enough x, the value returned by `primepi2()` (page 867) essentially amounts to a small perturbation of the value returned by `riemannr()` (page 868):

```
>>> primepi2(10**100)
[4.3619719871407024816e+97, 4.3619719871407032404e+97]
>>> riemannr(10**100)
4.3619719871407e+97
```

riemannr()

mpmath.**riemannr**(*x*)

Evaluates the Riemann R function, a smooth approximation of the prime counting function $\pi(x)$ (see `primepi()` (page 867)). The Riemann R function gives a fast numerical approximation useful e.g. to roughly estimate the number of primes in a given interval.

The Riemann R function is computed using the rapidly convergent Gram series,

$$R(x) = 1 + \sum_{k=1}^{\infty} \frac{\log^k x}{k k! \zeta(k+1)}.$$

From the Gram series, one sees that the Riemann R function is a well-defined analytic function (except for a branch cut along the negative real half-axis); it can be evaluated for arbitrary real or complex arguments.

The Riemann R function gives a very accurate approximation of the prime counting function. For example, it is wrong by at most 2 for $x < 1000$, and for $x = 10^9$ differs from the exact value of $\pi(x)$ by 79, or less than two parts in a million. It is about 10 times more accurate than the logarithmic integral estimate (see `li()` (page 705)), which however is even faster to evaluate. It is orders of magnitude more accurate than the extremely fast $x/\log x$ estimate.

Examples

For small arguments, the Riemann R function almost exactly gives the prime counting function if rounded to the nearest integer:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> primepi(50), riemannr(50)
(15, 14.9757023241462)
>>> max(abs(primepi(n)-int(round(riemannr(n)))) for n in range(100))
1
>>> max(abs(primepi(n)-int(round(riemannr(n)))) for n in range(300))
2
```

The Riemann R function can be evaluated for arguments far too large for exact determination of $\pi(x)$ to be computationally feasible with any presently known algorithm:

```
>>> riemannr(10**30)
1.46923988977204e+28
>>> riemannr(10**100)
4.3619719871407e+97
>>> riemannr(10**1000)
4.3448325764012e+996
```

A comparison of the Riemann R function and logarithmic integral estimates for $\pi(x)$ using exact values of $\pi(10^n)$ up to $n = 9$. The fractional error is shown in parentheses:

```
>>> exact = [4,25,168,1229,9592,78498,664579,5761455,50847534]
>>> for n, p in enumerate(exact):
...     n += 1
...     r, l = riemannr(10**n), li(10**n)
...     rerr, lerr = nstr((r-p)/p,3), nstr((l-p)/p,3)
...     print("%i %i %s(%s) %s(%s)" % (n, p, r, rerr, l, lerr))
...
1 4 4.56458314100509(0.141) 6.1655995047873(0.541)
2 25 25.6616332669242(0.0265) 30.1261415840796(0.205)
3 168 168.359446281167(0.00214) 177.609657990152(0.0572)
4 1229 1226.93121834343(-0.00168) 1246.13721589939(0.0139)
5 9592 9587.43173884197(-0.000476) 9629.8090010508(0.00394)
6 78498 78527.3994291277(0.000375) 78627.5491594622(0.00165)
7 664579 664667.447564748(0.000133) 664918.405048569(0.000511)
8 5761455 5761551.86732017(1.68e-5) 5762209.37544803(0.000131)
9 50847534 50847455.4277214(-1.55e-6) 50849234.9570018(3.35e-5)
```

The derivative of the Riemann R function gives the approximate probability for a number of magnitude x to be prime:

```
>>> diff(riemannr, 1000)
0.141903028110784
>>> mpf(primepi(1050) - primepi(950)) / 100
0.15
```

Evaluation is supported for arbitrary arguments and at arbitrary precision:

```
>>> mp.dps = 30
>>> riemannr(7.5)
3.72934743264966261918857135136
>>> riemannr(-4+2j)
(-0.551002208155486427591793957644 + 2.16966398138119450043195899746j)
```

Cyclotomic polynomials

cyclotomic()

mpmath.**cyclotomic**(*n*, *x*)

Evaluates the cyclotomic polynomial $\Phi_n(x)$, defined by

$$\Phi_n(x) = \prod_{\zeta} (x - \zeta)$$

where ζ ranges over all primitive n-th roots of unity (see `unitroots()` (page 660)). An equivalent representation, used for computation, is

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)} = \Phi_n(x)$$

where $\mu(m)$ denotes the Moebius function. The cyclotomic polynomials are integer polynomials, the first of which can be written explicitly as

$$\Phi_0(x) = 1$$
$$\Phi_1(x) = x - 1$$
$$\Phi_2(x) = x + 1$$
$$\Phi_3(x) = x^3 + x^2 + 1$$
$$\Phi_4(x) = x^2 + 1$$
$$\Phi_5(x) = x^4 + x^3 + x^2 + x + 1$$
$$\Phi_6(x) = x^2 - x + 1$$

Examples

The coefficients of low-order cyclotomic polynomials can be recovered using Taylor expansion:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(9):
...     p = chop(taylor(lambda x: cyclotomic(n,x), 0, 10))
...     print("%s %s" % (n, nstr(p[:10+1-p[::-1].index(1)])))
...
0 [1.0]
1 [-1.0, 1.0]
2 [1.0, 1.0]
3 [1.0, 1.0, 1.0]
4 [1.0, 0.0, 1.0]
5 [1.0, 1.0, 1.0, 1.0, 1.0]
6 [1.0, -1.0, 1.0]
7 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
8 [1.0, 0.0, 0.0, 0.0, 1.0]
```

The definition as a product over primitive roots may be checked by computing the product explicitly (for a real argument, this method will generally introduce numerical noise in the imaginary part):

```
>>> mp.dps = 25
>>> z = 3+4j
>>> cyclotomic(10, z)
(-419.0 - 360.0j)
>>> fprod(z-r for r in unitroots(10, primitive=True))
(-419.0 - 360.0j)
>>> z = 3
```

```
>>> cyclotomic(10, z)
61.0
>>> fprod(z-r for r in unitroots(10, primitive=True))
(61.0 - 3.14604560508856860705545 4e-25j)
```

Up to permutation, the roots of a given cyclotomic polynomial can be checked to agree with the list of primitive roots:

```
>>> p = taylor(lambda x: cyclotomic(6,x), 0, 6)[:3]
>>> for r in polyroots(p[::-1]):
...     print(r)
...
(0.5 - 0.866025403784438646763723 2j)
(0.5 + 0.866025403784438646763723 2j)
>>>
>>> for r in unitroots(6, primitive=True):
...     print(r)
...
(0.5 + 0.866025403784438646763723 2j)
(0.5 - 0.866025403784438646763723 2j)
```

Arithmetic functions

mangoldt()

mpmath.**mangoldt**(n)

Evaluates the von Mangoldt function $\Lambda(n) = \log p$ if $n = p^k$ a power of a prime, and $\Lambda(n) = 0$ otherwise.

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> [mangoldt(n) for n in range(-2,3)]
[0.0, 0.0, 0.0, 0.0, 0.6931471805599453094172321]
>>> mangoldt(6)
0.0
>>> mangoldt(7)
1.9459101490553133051053 53
>>> mangoldt(8)
0.6931471805599453094172321
>>> fsum(mangoldt(n) for n in range(101))
94.04531122935739224600493
>>> fsum(mangoldt(n) for n in range(10001))
10013.39669326311478372032
```

q-functions

q-Pochhammer symbol

qp()

mpmath.**qp**(*a, q=None, n=None, **kwargs*)

Evaluates the q-Pochhammer symbol (or q-rising factorial)

$$(a;q)_n = \prod_{k=0}^{n-1}(1-aq^k)$$

where $n=\infty$ is permitted if $|q|<1$. Called with two arguments, `qp(a,q)` computes $(a;q)_\infty$; with a single argument, `qp(q)` computes $(q;q)_\infty$. The special case

$$\phi(q) = (q;q)_\infty = \prod_{k=1}^{\infty}(1-q^k) = \sum_{k=-\infty}^{\infty}(-1)^k q^{(3k^2-k)/2}$$

is also known as the Euler function, or (up to a factor $q^{-1/24}$) the Dirichlet eta function.

Examples

If n is a positive integer, the function amounts to a finite product:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qp(2,3,5)
-725305.0
>>> fprod(1-2*3**k for k in range(5))
-725305.0
>>> qp(2,3,0)
1.0
```

Complex arguments are allowed:

```
>>> qp(2-1j, 0.75j)
(0.4628842231660149089976379 + 4.481821753552703090628793j)
```

The regular Pochhammer symbol $(a)_n$ is obtained in the following limit as $q \to 1$:

```
>>> a, n = 4, 7
>>> limit(lambda q: qp(q**a,q,n) / (1-q)**n, 1)
604800.0
>>> rf(a,n)
604800.0
```

The Taylor series of the reciprocal Euler function gives the partition function $P(n)$, i.e. the number of ways of writing n as a sum of positive integers:

```
>>> taylor(lambda q: 1/qp(q), 0, 10)
[1.0, 1.0, 2.0, 3.0, 5.0, 7.0, 11.0, 15.0, 22.0, 30.0, 42.0]
```

Special values include:

```
>>> qp(0)
1.0
>>> findroot(diffun(qp), -0.4)   # location of maximum
-0.411248479177954773444025 7
>>> qp(_)
1.228348867038575112586878
```

The q-Pochhammer symbol is related to the Jacobi theta functions. For example, the following identity holds:

```
>>> q = mpf(0.5)    # arbitrary
>>> qp(q)
0.28878809508660242127889 97
>>> root(3,-2)*root(q,-24)*jtheta(2,pi/6,root(q,6))
0.28878809508660242127889 97
```

q-gamma and factorial

qgamma()

mpmath.**qgamma**(*z, q, **kwargs*)

Evaluates the q-gamma function

$$\Gamma_q(z) = \frac{(q;q)_\infty}{(q^z;q)_\infty}(1-q)^{1-z}.$$

Examples

Evaluation for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qgamma(4,0.75)
4.046875
>>> qgamma(6,6)
121226245.0
>>> qgamma(3+4j, 0.5j)
(0.16630823822551998346300 88 + 0.01952474576025952984418217j)
```

The q-gamma function satisfies a functional equation similar to that of the ordinary gamma function:

```
>>> q = mpf(0.25)
>>> z = mpf(2.5)
>>> qgamma(z+1,q)
1.4282774248237609546859 12
>>> (1-q**z)/(1-q)*qgamma(z,q)
1.4282774248237609546859 12
```

qfac()

mpmath.**qfac**(*z, q, **kwargs*)

Evaluates the q-factorial,

$$[n]_q! = (1+q)(1+q+q^2)\cdots(1+q+\cdots+q^{n-1})$$

or more generally

$$[z]_q! = \frac{(q;q)_z}{(1-q)^z}.$$

Examples

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qfac(0,0)
1.0
```

```
>>> qfac(4,3)
2080.0
>>> qfac(5,6)
121226245.0
>>> qfac(1+1j, 2+1j)
(0.437055655132267247861369 5 + 0.260973983921603920370892 1j)
```

Hypergeometric q-series

qhyper()

mpmath.**qhyper**(*a_s, b_s, q, z, **kwargs*)

Evaluates the basic hypergeometric series or hypergeometric q-series

$$ {}_r\phi_s \left[\begin{matrix} a_1 & a_2 & \dots & a_r \\ b_1 & b_2 & \dots & b_s \end{matrix}; q, z\right] = \sum_{n=0}^{\infty} \frac{(a_1;q)_n, \dots, (a_r;q)_n}{(b_1;q)_n, \dots, (b_s;q)_n} \left((-1)^n q^{\binom{n}{2}}\right)^{1+s-r} \frac{z^n}{(q;q)_n} $$

where $(a;q)_n$ denotes the q-Pochhammer symbol (see qp() (page 871)).

Examples

Evaluation works for real and complex arguments:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> qhyper([0.5], [2.25], 0.25, 4)
-0.19758490912633560095343 85
>>> qhyper([0.5], [2.25], 0.25-0.25j, 4)
(2.806330244925716649839237 + 3.568997623337943121769938j)
>>> qhyper([1+j], [2,3+0.5j], 0.25, 3+4j)
(9.112885171773400017270226 - 1.272756997166375050700388j)
```

Comparing with a summation of the defining series, using nsum() (page 882):

```
>>> b, q, z = 3, 0.25, 0.5
>>> qhyper([], [b], q, z)
0.622113674825449558322832 4
>>> nsum(lambda n: z**n / qp(q,q,n)/qp(b,q,n) * q**(n*(n-1)), [0,inf])
0.622113674825449558322832 4
```

Numerical calculus

Polynomials

See also `taylor()` and `chebyfit()` for approximation of functions by polynomials.

Polynomial evaluation (polyval)

mpmath.**polyval**(*ctx, coeffs, x, derivative=False*)

Given coefficients $[c_n, \dots, c_2, c_1, c_0]$ and a number x, polyval() (page 874) evaluates the polynomial

$$ P(x) = c_n x^n + \dots + c_2 x^2 + c_1 x + c_0. $$

If *derivative=True* is set, polyval() (page 874) simultaneously evaluates $P(x)$ with the derivative, $P'(x)$, and returns the tuple $(P(x), P'(x))$.

```
>>> from mpmath import *
>>> mp.pretty = True
>>> polyval([3, 0, 2], 0.5)
2.75
>>> polyval([3, 0, 2], 0.5, derivative=True)
(2.75, 3.0)
```

The coefficients and the evaluation point may be any combination of real or complex numbers.

Polynomial roots (polyroots)

mpmath.**polyroots**(*ctx, coeffs, maxsteps=50, cleanup=True, extraprec=10, error=False*)

Computes all roots (real or complex) of a given polynomial. The roots are returned as a sorted list, where real roots appear first followed by complex conjugate roots as adjacent elements. The polynomial should be given as a list of coefficients, in the format used by polyval() (page 874). The leading coefficient must be nonzero.

With *error=True*, polyroots() (page 875) returns a tuple *(roots, err)* where *err* is an estimate of the maximum error among the computed roots.

Examples

Finding the three real roots of $x^3 - x^2 - 14x + 24$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(polyroots([1,-1,-14,24]), 4)
[-4.0, 2.0, 3.0]
```

Finding the two complex conjugate roots of $4x^2 + 3x + 2$, with an error estimate:

```
>>> roots, err = polyroots([4,3,2], error=True)
>>> for r in roots:
...     print(r)
...
(-0.375 + 0.59947894041409j)
(-0.375 - 0.59947894041409j)
>>>
>>> err
2.22044604925031e-16
>>>
>>> polyval([4,3,2], roots[0])
(2.22044604925031e-16 + 0.0j)
>>> polyval([4,3,2], roots[1])
(2.22044604925031e-16 + 0.0j)
```

The following example computes all the 5th roots of unity; that is, the roots of $x^5 - 1$:

```
>>> mp.dps = 20
>>> for r in polyroots([1, 0, 0, 0, 0, -1]):
...     print(r)
...
1.0
(-0.8090169943749474241 + 0.58778525229247312917j)
(-0.8090169943749474241 - 0.58778525229247312917j)
(0.3090169943749474241 + 0.95105651629515357212j)
(0.3090169943749474241 - 0.95105651629515357212j)
```

Precision and conditioning

Provided there are no repeated roots, `polyroots()` (page 875) can typically compute all roots of an arbitrary polynomial to high precision:

```
>>> mp.dps = 60
>>> for r in polyroots([1, 0, -10, 0, 1]):
...     print r
...
-3.14626436994197234232913506571557044551247712918732870123249
-0.317837245195782244725757617296174288373133378433432554879127
0.317837245195782244725757617296174288373133378433432554879127
3.14626436994197234232913506571557044551247712918732870123249
>>>
>>> sqrt(3) + sqrt(2)
3.14626436994197234232913506571557044551247712918732870123249
>>> sqrt(3) - sqrt(2)
0.317837245195782244725757617296174288373133378433432554879127
```

Algorithm

`polyroots()` (page 875) implements the Durand-Kerner method [1], which uses complex arithmetic to locate all roots simultaneously. The Durand-Kerner method can be viewed as approximately performing simultaneous Newton iteration for all the roots. In particular, the convergence to simple roots is quadratic, just like Newton's method.

Although all roots are internally calculated using complex arithmetic, any root found to have an imaginary part smaller than the estimated numerical error is truncated to a real number. Real roots are placed first in the returned list, sorted by value. The remaining complex roots are sorted by real their parts so that conjugate roots end up next to each other.

References

1. http://en.wikipedia.org/wiki/Durand-Kerner_method

Root-finding and optimization

Root-finding (`findroot`)

`mpmath.findroot(f, x0, solver=Secant, tol=None, verbose=False, verify=True, **kwargs)`

Find a solution to $f(x) = 0$, using *x0* as starting point or interval for *x*.

Multidimensional overdetermined systems are supported. You can specify them using a function or a list of functions.

If the found root does not satisfy $|f(x)^2 < \text{tol}|$, an exception is raised (this can be disabled with *verify=False*).

Arguments

f one dimensional function

x0 starting point, several starting points or interval (depends on solver)

tol the returned solution has an error smaller than this

verbose print additional information for each iteration if true

verify verify the solution and raise a ValueError if $|f(x) > \text{tol}|$

solver a generator for *f* and *x0* returning approximative solution and error

maxsteps after how many steps the solver will cancel

df first derivative of f (used by some solvers)

d2f second derivative of f (used by some solvers)

multidimensional force multidimensional solving

J Jacobian matrix of f (used by multidimensional solvers)

norm used vector norm (used by multidimensional solvers)

solver has to be callable with `(f, x0, **kwargs)` and return an generator yielding pairs of approximative solution and estimated error (which is expected to be positive). You can use the following string aliases: 'secant', 'mnewton', 'halley', 'muller', 'illinois', 'pegasus', 'anderson', 'ridder', 'anewton', 'bisect'

See mpmath.optimization for their documentation.

Examples

The function `findroot()` (page 876) locates a root of a given function using the secant method by default. A simple example use of the secant method is to compute π as the root of $\sin x$ closest to $x_0 = 3$:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> findroot(sin, 3)
3.14159265358979323846264338328
```

The secant method can be used to find complex roots of analytic functions, although it must in that case generally be given a nonreal starting value (or else it will never leave the real line):

```
>>> mp.dps = 15
>>> findroot(lambda x: x**3 + 2*x + 1, j)
(0.226698825758202 + 1.46771150871022j)
```

A nice application is to compute nontrivial roots of the Riemann zeta function with many digits (good initial values are needed for convergence):

```
>>> mp.dps = 30
>>> findroot(zeta, 0.5+14j)
(0.5 + 14.1347251417346937904572519836j)
```

The secant method can also be used as an optimization algorithm, by passing it a derivative of a function. The following example locates the positive minimum of the gamma function:

```
>>> mp.dps = 20
>>> findroot(lambda x: diff(gamma, x), 1)
1.4616321449683623413
```

Finally, a useful application is to compute inverse functions, such as the Lambert W function which is the inverse of we^w, given the first term of the solution's asymptotic expansion as the initial value. In basic cases, this gives identical results to mpmath's built-in `lambertw` function:

```
>>> def lambert(x):
...     return findroot(lambda w: w*exp(w) - x, log(1+x))
...
>>> mp.dps = 15
>>> lambert(1); lambertw(1)
```

```
0.567143290409784
0.567143290409784
>>> lambert(1000); lambert(1000)
5.2496028524016
5.2496028524016
```

Multidimensional functions are also supported:

```
>>> f = [lambda x1, x2: x1**2 + x2,
...      lambda x1, x2: 5*x1**2 - 3*x1 + 2*x2 - 3]
>>> findroot(f, (0, 0))
[-0.618033988749895]
[-0.381966011250105]
>>> findroot(f, (10, 10))
[ 1.61803398874989]
[-2.61803398874989]
```

You can verify this by solving the system manually.

Please note that the following (more general) syntax also works:

```
>>> def f(x1, x2):
...     return x1**2 + x2, 5*x1**2 - 3*x1 + 2*x2 - 3
...
>>> findroot(f, (0, 0))
[-0.618033988749895]
[-0.381966011250105]
```

Multiple roots

For multiple roots all methods of the Newtonian family (including secant) converge slowly. Consider this example:

```
>>> f = lambda x: (x - 1)**99
>>> findroot(f, 0.9, verify=False)
0.918073542444929
```

Even for a very close starting point the secant method converges very slowly. Use `verbose=True` to illustrate this.

It is possible to modify Newton's method to make it converge regardless of the root's multiplicity:

```
>>> findroot(f, -10, solver='mnewton')
1.0
```

This variant uses the first and second derivative of the function, which is not very efficient.

Alternatively you can use an experimental Newtonian solver that keeps track of the speed of convergence and accelerates it using Steffensen's method if necessary:

```
>>> findroot(f, -10, solver='anewton', verbose=True)
x:     -9.88888888888888888889
error: 0.11111111111111111111
converging slowly
x:     -9.77890011223344556678
error: 0.10998877665544332211
converging slowly
x:     -9.67002233332199662166
error: 0.10887778911448945119
```

```
converging slowly
accelerating convergence
x:     -9.562244329955107766 9
error: 0.10777800336688885476 4
converging slowly
x:     0.99999999999999999214
error: 10.562244329955107759
x:     1.0
error: 7.8598304758094664213e-18
1.0
```

Complex roots

For complex roots it's recommended to use Muller's method as it converges even for real starting points very fast:

```
>>> findroot(lambda x: x**4 + x + 1, (0, 1, 2), solver='muller')
(0.727136084491197 + 0.934099289460529j)
```

Intersection methods

When you need to find a root in a known interval, it's highly recommended to use an intersection-based solver like `'anderson'` or `'ridder'`. Usually they converge faster and more reliable. They have however problems with multiple roots and usually need a sign change to find a root:

```
>>> findroot(lambda x: x**3, (-1, 1), solver='anderson')
0.0
```

Be careful with symmetric functions:

```
>>> findroot(lambda x: x**2, (-1, 1), solver='anderson')
Traceback (most recent call last):
  ...
ZeroDivisionError
```

It fails even for better starting points, because there is no sign change:

```
>>> findroot(lambda x: x**2, (-1, .5), solver='anderson')
Traceback (most recent call last):
  ...
ValueError: Could not find root within given tolerance. (1 > 2.1684e-19)
Try another starting point or tweak arguments.
```

Solvers

class `mpmath.calculus.optimization.`**`Secant`**(*ctx, f, x0, **kwargs*)

1d-solver generating pairs of approximative root and error.

Needs starting points x0 and x1 close to the root. x1 defaults to x0 + 0.25.

Pro:

- converges fast

Contra:

- converges slowly for multiple roots

class `mpmath.calculus.optimization.`**`Newton`**(*ctx, f, x0, **kwargs*)

1d-solver generating pairs of approximative root and error.

Needs starting points x0 close to the root.

Pro:

- converges fast
- sometimes more robust than secant with bad second starting point

Contra:

- converges slowly for multiple roots
- needs first derivative
- 2 function evaluations per iteration

class `mpmath.calculus.optimization.`**`MNewton`**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Needs starting point x0 close to the root. Uses modified Newton's method that converges fast regardless of the multiplicity of the root.

Pro:

- converges fast for multiple roots

Contra:

- needs first and second derivative of f
- 3 function evaluations per iteration

class `mpmath.calculus.optimization.`**`Halley`**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Needs a starting point x0 close to the root. Uses Halley's method with cubic convergence rate.

Pro:

- converges even faster the Newton's method
- useful when computing with *many* digits

Contra:

- needs first and second derivative of f
- 3 function evaluations per iteration
- converges slowly for multiple roots

class `mpmath.calculus.optimization.`**`Muller`**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Needs starting points x0, x1 and x2 close to the root. x1 defaults to x0 + 0.25; x2 to x1 + 0.25. Uses Muller's method that converges towards complex roots.

Pro:

- converges fast (somewhat faster than secant)
- can find complex roots

Contra:

- converges slowly for multiple roots
- may have complex values for real starting points and real roots

http://en.wikipedia.org/wiki/Muller's_method

class `mpmath.calculus.optimization.`**`Bisection`**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Uses bisection method to find a root of f in [a, b]. Might fail for multiple roots (needs sign change).

Pro:

- robust and reliable

Contra:

- converges slowly
- needs sign change

class `mpmath.calculus.optimization.`**`Illinois`**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Uses Illinois method or similar to find a root of f in [a, b]. Might fail for multiple roots (needs sign change). Combines bisect with secant (improved regula falsi).

The only difference between the methods is the scaling factor m, which is used to ensure convergence (you can choose one using the 'method' keyword):

Illinois method ('illinois'): m = 0.5

Pegasus method ('pegasus'): m = fb/(fb + fz)

Anderson-Bjoerk method ('anderson'): m = 1 - fz/fb if positive else 0.5

Pro:

- converges very fast

Contra:

- has problems with multiple roots
- needs sign change

class `mpmath.calculus.optimization.`**`Pegasus`**

1d-solver generating pairs of approximative root and error.

Uses Pegasus method to find a root of f in [a, b]. Wrapper for illinois to use method='pegasus'.

class `mpmath.calculus.optimization.`**`Anderson`**

1d-solver generating pairs of approximative root and error.

Uses Anderson-Bjoerk method to find a root of f in [a, b]. Wrapper for illinois to use method='pegasus'.

class `mpmath.calculus.optimization.`**`Ridder`**(*ctx*, *f*, *x0*, ***kwargs*)

1d-solver generating pairs of approximative root and error.

Ridders' method to find a root of f in [a, b]. Is told to perform as well as Brent's method while being simpler.

Pro:

- very fast
- simpler than Brent's method

Contra:

- two function evaluations per step
- has problems with multiple roots

- needs sign change

http://en.wikipedia.org/wiki/Ridders'_method

class mpmath.calculus.optimization.**ANewton**(*ctx*, *f*, *x0*, ***kwargs*)

EXPERIMENTAL 1d-solver generating pairs of approximative root and error.

Uses Newton's method modified to use Steffensens method when convergence is slow. (I.e. for multiple roots.)

class mpmath.calculus.optimization.**MDNewton**(*ctx*, *f*, *x0*, ***kwargs*)

Find the root of a vector function numerically using Newton's method.

f is a vector function representing a nonlinear equation system.

x0 is the starting point close to the root.

J is a function returning the Jacobian matrix for a point.

Supports overdetermined systems.

Use the 'norm' keyword to specify which norm to use. Defaults to max-norm. The function to calculate the Jacobian matrix can be given using the keyword 'J'. Otherwise it will be calculated numerically.

Please note that this method converges only locally. Especially for high- dimensional systems it is not trivial to find a good starting point being close enough to the root.

It is recommended to use a faster, low-precision solver from SciPy [1] or OpenOpt [2] to get an initial guess. Afterwards you can use this method for root-polishing to any precision.

[1] http://scipy.org

[2] http://openopt.org

Sums, products, limits and extrapolation

The functions listed here permit approximation of infinite sums, products, and other sequence limits. Use `mpmath.fsum()` (page 639) and `mpmath.fprod()` (page 639) for summation and multiplication of finite sequences.

Summation

nsum()

mpmath.**nsum**(*ctx*, *f*, **intervals*, ***options*)

Computes the sum

$$S = \sum_{k=a}^{b} f(k)$$

where (a, b) = *interval*, and where $a = -\infty$ and/or $b = \infty$ are allowed, or more generally

$$S = \sum_{k_1=a_1}^{b_1} \cdots \sum_{k_n=a_n}^{b_n} f(k_1, \ldots, k_n)$$

if multiple intervals are given.

Two examples of infinite series that can be summed by nsum() (page 882), where the first converges rapidly and the second converges slowly, are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nsum(lambda n: 1/fac(n), [0, inf])
2.71828182845905
>>> nsum(lambda n: 1/n**2, [1, inf])
1.64493406684823
```

When appropriate, nsum() (page 882) applies convergence acceleration to accurately estimate the sums of slowly convergent series. If the series is finite, nsum() (page 882) currently does not attempt to perform any extrapolation, and simply calls fsum() (page 639).

Multidimensional infinite series are reduced to a single-dimensional series over expanding hypercubes; if both infinite and finite dimensions are present, the finite ranges are moved innermost. For more advanced control over the summation order, use nested calls to nsum() (page 882), or manually rewrite the sum as a single-dimensional series.

Options

tol Desired maximum final error. Defaults roughly to the epsilon of the working precision.

method Which summation algorithm to use (described below). Default: 'richardson+shanks'.

maxterms Cancel after at most this many terms. Default: 10*dps.

steps An iterable giving the number of terms to add between each extrapolation attempt. The default sequence is [10, 20, 30, 40, ...]. For example, if you know that approximately 100 terms will be required, efficiency might be improved by setting this to [100, 10]. Then the first extrapolation will be performed after 100 terms, the second after 110, etc.

verbose Print details about progress.

ignore If enabled, any term that raises ArithmeticError or ValueError (e.g. through division by zero) is replaced by a zero. This is convenient for lattice sums with a singular term near the origin.

Methods

Unfortunately, an algorithm that can efficiently sum any infinite series does not exist. nsum() (page 882) implements several different algorithms that each work well in different cases. The *method* keyword argument selects a method.

The default method is 'r+s', i.e. both Richardson extrapolation and Shanks transformation is attempted. A slower method that handles more cases is 'r+s+e'. For very high precision summation, or if the summation needs to be fast (for example if multiple sums need to be evaluated), it is a good idea to investigate which one method works best and only use that.

'richardson' / 'r': Uses Richardson extrapolation. Provides useful extrapolation when $f(k) \sim P(k)/Q(k)$ or when $f(k) \sim (-1)^k P(k)/Q(k)$ for polynomials P and Q. See richardson() (page 893) for additional information.

'shanks' / 's': Uses Shanks transformation. Typically provides useful extrapolation when $f(k) \sim c^k$ or when successive terms alternate signs. Is able to sum some divergent series. See shanks() (page 894) for additional information.

'euler-maclaurin' / 'e': Uses the Euler-Maclaurin summation formula to approximate the remainder sum by an integral. This requires high-order numerical derivatives and numerical integration. The advantage of this algorithm is that it works regardless of the decay rate of f, as long as f is sufficiently smooth. See sumem() (page 888) for additional information.

'direct' / 'd': Does not perform any extrapolation. This can be used (and should only be used for) rapidly convergent series. The summation automatically stops when the terms decrease below the target tolerance.

Basic examples

A finite sum:

```
>>> nsum(lambda k: 1/k, [1, 6])
2.45
```

Summation of a series going to negative infinity and a doubly infinite series:

```
>>> nsum(lambda k: 1/k**2, [-inf, -1])
1.64493406684823
>>> nsum(lambda k: 1/(1+k**2), [-inf, inf])
3.15334809493716
```

nsum() (page 882) handles sums of complex numbers:

```
>>> nsum(lambda k: (0.5+0.25j)**k, [0, inf])
(1.6 + 0.8j)
```

The following sum converges very rapidly, so it is most efficient to sum it by disabling convergence acceleration:

```
>>> mp.dps = 1000
>>> a = nsum(lambda k: -(-1)**k * k**2 / fac(2*k), [1, inf],
...     method='direct')
>>> b = (cos(1)+sin(1))/4
>>> abs(a-b) < mpf('1e-998')
True
```

Examples with Richardson extrapolation

Richardson extrapolation works well for sums over rational functions, as well as their alternating counterparts:

```
>>> mp.dps = 50
>>> nsum(lambda k: 1 / k**3, [1, inf],
...     method='richardson')
1.2020569031595942853997381615114499907649862923405
>>> zeta(3)
1.2020569031595942853997381615114499907649862923405

>>> nsum(lambda n: (n + 3)/(n**3 + n**2), [1, inf],
...     method='richardson')
2.9348022005446793094172454999380755676568497036204
>>> pi**2/2-2
2.9348022005446793094172454999380755676568497036204

>>> nsum(lambda k: (-1)**k / k**3, [1, inf],
...     method='richardson')
-0.90154267736969571404980362113358749307373971925537
```

```
>>> -3*zeta(3)/4
-0.901542677369695714049803621133587493073739719255 38
```

Examples with Shanks transformation

The Shanks transformation works well for geometric series and typically provides excellent acceleration for Taylor series near the border of their disk of convergence. Here we apply it to a series for $\log(2)$, which can be seen as the Taylor series for $\log(1+x)$ with $x=1$:

```
>>> nsum(lambda k: -(-1)**k/k, [1, inf],
...     method='shanks')
0.693147180559945309417232121458176568075500134360 25
>>> log(2)
0.693147180559945309417232121458176568075500134360 25
```

Here we apply it to a slowly convergent geometric series:

```
>>> nsum(lambda k: mpf('0.995')**k, [0, inf],
...     method='shanks')
200.0
```

Finally, Shanks' method works very well for alternating series where $f(k) = (-1)^k g(k)$, and often does so regardless of the exact decay rate of $g(k)$:

```
>>> mp.dps = 15
>>> nsum(lambda k: (-1)**(k+1) / k**1.5, [1, inf],
...     method='shanks')
0.765147024625408
>>> (2-sqrt(2))*zeta(1.5)/2
0.765147024625408
```

The following slowly convergent alternating series has no known closed-form value. Evaluating the sum a second time at higher precision indicates that the value is probably correct:

```
>>> nsum(lambda k: (-1)**k / log(k), [2, inf],
...     method='shanks')
0.924299897222939
>>> mp.dps = 30
>>> nsum(lambda k: (-1)**k / log(k), [2, inf],
...     method='shanks')
0.924299897222938855959570181 36
```

Examples with Euler-Maclaurin summation

The sum in the following example has the wrong rate of convergence for either Richardson or Shanks to be effective.

```
>>> f = lambda k: log(k)/k**2.5
>>> mp.dps = 15
>>> nsum(f, [1, inf], method='euler-maclaurin')
0.38734195032621
>>> -diff(zeta, 2.5)
0.38734195032621
```

Increasing `steps` improves speed at higher precision:

```
>>> mp.dps = 50
>>> nsum(f, [1, inf], method='euler-maclaurin', steps=[250])
0.38734195032620997271199237593105101319948228874688
```

```
>>> -diff(zeta, 2.5)
0.3873419503262099727119923759310510131994822887468 8
```

Divergent series

The Shanks transformation is able to sum some *divergent* series. In particular, it is often able to sum Taylor series beyond their radius of convergence (this is due to a relation between the Shanks transformation and Pade approximations; see `pade()` (page 912) for an alternative way to evaluate divergent Taylor series).

Here we apply it to $\log(1+x)$ far outside the region of convergence:

```
>>> mp.dps = 50
>>> nsum(lambda k: -(-9)**k/k, [1, inf],
...     method='shanks')
2.3025850929940456840179914546843642076011014886288
>>> log(10)
2.3025850929940456840179914546843642076011014886288
```

A particular type of divergent series that can be summed using the Shanks transformation is geometric series. The result is the same as using the closed-form formula for an infinite geometric series:

```
>>> mp.dps = 15
>>> for n in range(-8, 8):
...     if n == 1:
...         continue
...     print("%s %s %s" % (mpf(n), mpf(1)/(1-n),
...         nsum(lambda k: n**k, [0, inf], method='shanks')))
...
-8.0 0.111111111111111 0.111111111111111
-7.0 0.125 0.125
-6.0 0.142857142857143 0.142857142857143
-5.0 0.166666666666667 0.166666666666667
-4.0 0.2 0.2
-3.0 0.25 0.25
-2.0 0.333333333333333 0.333333333333333
-1.0 0.5 0.5
0.0 1.0 1.0
2.0 -1.0 -1.0
3.0 -0.5 -0.5
4.0 -0.333333333333333 -0.333333333333333
5.0 -0.25 -0.25
6.0 -0.2 -0.2
7.0 -0.166666666666667 -0.166666666666667
```

Multidimensional sums

Any combination of finite and infinite ranges is allowed for the summation indices:

```
>>> mp.dps = 15
>>> nsum(lambda x,y: x+y, [2,3], [4,5])
28.0
>>> nsum(lambda x,y: x/2**y, [1,3], [1,inf])
6.0
>>> nsum(lambda x,y: y/2**x, [1,inf], [1,3])
6.0
>>> nsum(lambda x,y,z: z/(2**x*2**y), [1,inf], [1,inf], [3,4])
7.0
>>> nsum(lambda x,y,z: y/(2**x*2**z), [1,inf], [3,4], [1,inf])
```

```
7.0
>>> nsum(lambda x,y,z: x/(2**z*2**y), [3,4], [1,inf], [1,inf])
7.0
```

Some nice examples of double series with analytic solutions or reductions to single-dimensional series (see [1]):

```
>>> nsum(lambda m, n: 1/2**(m*n), [1,inf], [1,inf])
1.60669515241529
>>> nsum(lambda n: 1/(2**n-1), [1,inf])
1.60669515241529

>>> nsum(lambda i,j: (-1)**(i+j)/(i**2+j**2), [1,inf], [1,inf])
0.278070510848213
>>> pi*(pi-3*ln2)/12
0.278070510848213

>>> nsum(lambda i,j: (-1)**(i+j)/(i+j)**2, [1,inf], [1,inf])
0.129319852864168
>>> altzeta(2) - altzeta(1)
0.129319852864168

>>> nsum(lambda i,j: (-1)**(i+j)/(i+j)**3, [1,inf], [1,inf])
0.0790756439455825
>>> altzeta(3) - altzeta(2)
0.0790756439455825

>>> nsum(lambda m,n: m**2*n/(3**m*(n*3**m+m*3**n)),
...     [1,inf], [1,inf])
0.28125
>>> mpf(9)/32
0.28125

>>> nsum(lambda i,j: fac(i-1)*fac(j-1)/fac(i+j),
...     [1,inf], [1,inf], workprec=400)
1.64493406684823
>>> zeta(2)
1.64493406684823
```

A hard example of a multidimensional sum is the Madelung constant in three dimensions (see [2]). The defining sum converges very slowly and only conditionally, so `nsum()` (page 882) is lucky to obtain an accurate value through convergence acceleration. The second evaluation below uses a much more efficient, rapidly convergent 2D sum:

```
>>> nsum(lambda x,y,z: (-1)**(x+y+z)/(x*x+y*y+z*z)**0.5,
...     [-inf,inf], [-inf,inf], [-inf,inf], ignore=True)
-1.74756459463318
>>> nsum(lambda x,y: -12*pi*sech(0.5*pi * \
...     sqrt((2*x+1)**2+(2*y+1)**2))**2, [0,inf], [0,inf])
-1.74756459463318
```

Another example of a lattice sum in 2D:

```
>>> nsum(lambda x,y: (-1)**(x+y) / (x**2+y**2), [-inf,inf],
...     [-inf,inf], ignore=True)
-2.1775860903036
>>> -pi*ln2
-2.1775860903036
```

An example of an Eisenstein series:

```
>>> nsum(lambda m,n: (m+n*1j)**(-4), [-inf,inf], [-inf,inf],
...     ignore=True)
(3.1512120021539 + 0.0j)
```

References

1. [Weisstein] (page 1626) http://mathworld.wolfram.com/DoubleSeries.html,
2. [Weisstein] (page 1626) http://mathworld.wolfram.com/MadelungConstants.html

sumem()

mpmath.**sumem**(*ctx, f, interval, tol=None, reject=10, integral=None, adiffs=None, bdiffs=None, verbose=False, error=False, _fast_abort=False*)

Uses the Euler-Maclaurin formula to compute an approximation accurate to within `tol` (which defaults to the present epsilon) of the sum

$$S = \sum_{k=a}^{b} f(k)$$

where (a, b) are given by `interval` and a or b may be infinite. The approximation is

$$S \sim \int_a^b f(x)\,dx + \frac{f(a)+f(b)}{2} + \sum_{k=1}^{\infty} \frac{B_{2k}}{(2k)!}\left(f^{(2k-1)}(b) - f^{(2k-1)}(a)\right).$$

The last sum in the Euler-Maclaurin formula is not generally convergent (a notable exception is if f is a polynomial, in which case Euler-Maclaurin actually gives an exact result).

The summation is stopped as soon as the quotient between two consecutive terms falls below *reject*. That is, by default (*reject* = 10), the summation is continued as long as each term adds at least one decimal.

Although not convergent, convergence to a given tolerance can often be "forced" if $b = \infty$ by summing up to $a + N$ and then applying the Euler-Maclaurin formula to the sum over the range $(a + N + 1, \ldots, \infty)$. This procedure is implemented by `nsum()` (page 882).

By default numerical quadrature and differentiation is used. If the symbolic values of the integral and endpoint derivatives are known, it is more efficient to pass the value of the integral explicitly as `integral` and the derivatives explicitly as `adiffs` and `bdiffs`. The derivatives should be given as iterables that yield $f(a), f'(a), f''(a), \ldots$ (and the equivalent for b).

Examples

Summation of an infinite series, with automatic and symbolic integral and derivative values (the second should be much faster):

```
>>> from mpmath import *
>>> mp.dps = 50; mp.pretty = True
>>> sumem(lambda n: 1/n**2, [32, inf])
0.0317433665203020901265816804387414271413288641 3417
>>> I = mpf(1)/32
>>> D = adiffs=((-1)**n*fac(n+1)*32**(-2-n) for n in range(999))
>>> sumem(lambda n: 1/n**2, [32, inf], integral=I, adiffs=D)
0.0317433665203020901265816804387414271413288641 3417
```

An exact evaluation of a finite polynomial sum:

```
>>> sumem(lambda n: n**5-12*n**2+3*n, [-100000, 200000])
10500155000624963999742499550000.0
>>> print(sum(n**5-12*n**2+3*n for n in range(-100000, 200001)))
10500155000624963999742499550000
```

sumap()

mpmath.**sumap**(*ctx*, *f*, *interval*, *integral=None*, *error=False*)

Evaluates an infinite series of an analytic summand f using the Abel-Plana formula

$$\sum_{k=0}^{\infty} f(k) = \int_0^{\infty} f(t)dt + \frac{1}{2}f(0) + i\int_0^{\infty} \frac{f(it)-f(-it)}{e^{2\pi t}-1}dt.$$

Unlike the Euler-Maclaurin formula (see sumem() (page 888)), the Abel-Plana formula does not require derivatives. However, it only works when $|f(it)-f(-it)|$ does not increase too rapidly with t.

Examples

The Abel-Plana formula is particularly useful when the summand decreases like a power of k; for example when the sum is a pure zeta function:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> sumap(lambda k: 1/k**2.5, [1,inf])
1.341487257250917179756 77
>>> zeta(2.5)
1.341487257250917179756 77
>>> sumap(lambda k: 1/(k+1j)**(2.5+2.5j), [1,inf])
(-3.385361068546473342286084 - 0.7432082105196321803869551j)
>>> zeta(2.5+2.5j, 1+1j)
(-3.385361068546473342286084 - 0.7432082105196321803869551j)
```

If the series is alternating, numerical quadrature along the real line is likely to give poor results, so it is better to evaluate the first term symbolically whenever possible:

```
>>> n=3; z=-0.75
>>> I = expint(n,-log(z))
>>> chop(sumap(lambda k: z**k / k**n, [1,inf], integral=I))
-0.691703603690459451014144 8
>>> polylog(n,z)
-0.691703603690459451014144 8
```

Products

nprod()

mpmath.**nprod**(*ctx*, *f*, *interval*, *nsum=False*, ***kwargs*)

Computes the product

$$P = \prod_{k=a}^{b} f(k)$$

where (a, b) = *interval*, and where $a = -\infty$ and/or $b = \infty$ are allowed.

By default, nprod() (page 889) uses the same extrapolation methods as nsum() (page 882), except applied to the partial products rather than partial sums, and the same

keyword options as for nsum() (page 882) are supported. If nsum=True, the product is instead computed via nsum() (page 882) as

$$P = \exp\left(\sum_{k=a}^{b} \log(f(k))\right).$$

This is slower, but can sometimes yield better results. It is also required (and used automatically) when Euler-Maclaurin summation is requested.

Examples

A simple finite product:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> nprod(lambda k: k, [1, 4])
24.0
```

A large number of infinite products have known exact values, and can therefore be used as a reference. Most of the following examples are taken from MathWorld [1].

A few infinite products with simple values are:

```
>>> 2*nprod(lambda k: (4*k**2)/(4*k**2-1), [1, inf])
3.141592653589793238462643
>>> nprod(lambda k: (1+1/k)**2/(1+2/k), [1, inf])
2.0
>>> nprod(lambda k: (k**3-1)/(k**3+1), [2, inf])
0.6666666666666666666666667
>>> nprod(lambda k: (1-1/k**2), [2, inf])
0.5
```

Next, several more infinite products with more complicated values:

```
>>> nprod(lambda k: exp(1/k**2), [1, inf]); exp(pi**2/6)
5.180668317897115748416626
5.180668317897115748416626

>>> nprod(lambda k: (k**2-1)/(k**2+1), [2, inf]); pi*csch(pi)
0.272029054982133162950236 6
0.272029054982133162950236 6

>>> nprod(lambda k: (k**4-1)/(k**4+1), [2, inf])
0.848054049352900392129650 2
>>> pi*sinh(pi)/(cosh(sqrt(2)*pi)-cos(sqrt(2)*pi))
0.848054049352900392129650 2

>>> nprod(lambda k: (1+1/k+1/k**2)**2/(1+2/k+3/k**2), [1, inf])
1.848936182858244485224927
>>> 3*sqrt(2)*cosh(pi*sqrt(3)/2)**2*csch(pi*sqrt(2))/pi
1.848936182858244485224927

>>> nprod(lambda k: (1-1/k**4), [2, inf]); sinh(pi)/(4*pi)
0.919019477593744430173924 4
0.919019477593744430173924 4

>>> nprod(lambda k: (1-1/k**6), [2, inf])
0.982684277742192518324475 9
>>> (1+cosh(pi*sqrt(3)))/(12*pi**2)
0.982684277742192518324475 9
```

```
>>> nprod(lambda k: (1+1/k**2), [2, inf]); sinh(pi)/(2*pi)
1.83803895518748886034784 9
1.83803895518748886034784 9

>>> nprod(lambda n: (1+1/n)**n * exp(1/(2*n)-1), [1, inf])
1.44725592689036529895913 8
>>> exp(1+euler/2)/sqrt(2*pi)
1.44725592689036529895913 8
```

The following two products are equivalent and can be evaluated in terms of a Jacobi theta function. Pi can be replaced by any value (as long as convergence is preserved):

```
>>> nprod(lambda k: (1-pi**-k)/(1+pi**-k), [1, inf])
0.383845120748167240477868 6
>>> nprod(lambda k: tanh(k*log(pi)/2), [1, inf])
0.383845120748167240477868 6
>>> jtheta(4,0,1/pi)
0.383845120748167240477868 6
```

This product does not have a known closed form value:

```
>>> nprod(lambda k: (1-1/2**k), [1, inf])
0.288788095086602421278899 7
```

A product taken from $-\infty$:

```
>>> nprod(lambda k: 1-k**(-3), [-inf,-2])
0.809396597366290109578680 5
>>> cosh(pi*sqrt(3)/2)/(3*pi)
0.809396597366290109578680 5
```

A doubly infinite product:

```
>>> nprod(lambda k: exp(1/(1+k**2)), [-inf, inf])
23.4143268823186433742003 5
>>> exp(pi/tanh(pi))
23.4143268823186433742003 5
```

A product requiring the use of Euler-Maclaurin summation to compute an accurate value:

```
>>> nprod(lambda k: (1-1/k**2.5), [2, inf], method='e')
0.696155111336231052898125
```

References

1. [Weisstein] (page 1626) http://mathworld.wolfram.com/InfiniteProduct.html

Limits (limit)

limit()

mpmath.**limit**(*ctx, f, x, direction=1, exp=False, **kwargs*)

Computes an estimate of the limit

$$\lim_{t \to x} f(t)$$

where x may be finite or infinite.

For finite x, `limit()` (page 891) evaluates $f(x + d/n)$ for consecutive integer values of n, where the approach direction d may be specified using the *direction* keyword argument. For infinite x, `limit()` (page 891) evaluates values of $f(\text{sign}(x) \cdot n)$.

If the approach to the limit is not sufficiently fast to give an accurate estimate directly, `limit()` (page 891) attempts to find the limit using Richardson extrapolation or the Shanks transformation. You can select between these methods using the *method* keyword (see documentation of `nsum()` (page 882) for more information).

Options

The following options are available with essentially the same meaning as for `nsum()` (page 882): *tol*, *method*, *maxterms*, *steps*, *verbose*.

If the option *exp=True* is set, f will be sampled at exponentially spaced points $n = 2^1, 2^2, 2^3, \ldots$ instead of the linearly spaced points $n = 1, 2, 3, \ldots$. This can sometimes improve the rate of convergence so that `limit()` (page 891) may return a more accurate answer (and faster). However, do note that this can only be used if f supports fast and accurate evaluation for arguments that are extremely close to the limit point (or if infinite, very large arguments).

Examples

A basic evaluation of a removable singularity:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> limit(lambda x: (x-sin(x))/x**3, 0)
0.166666666666666666666666666667
```

Computing the exponential function using its limit definition:

```
>>> limit(lambda n: (1+3/n)**n, inf)
20.0855369231876677409285296546
>>> exp(3)
20.0855369231876677409285296546
```

A limit for π:

```
>>> f = lambda n: 2**(4*n+1)*fac(n)**4/(2*n+1)/fac(2*n)**2
>>> limit(f, inf)
3.14159265358979323846264338328
```

Calculating the coefficient in Stirling's formula:

```
>>> limit(lambda n: fac(n) / (sqrt(n)*(n/e)**n), inf)
2.50662827463100050241576528481
>>> sqrt(2*pi)
2.50662827463100050241576528481
```

Evaluating Euler's constant γ using the limit representation

$$\gamma = \lim_{n\to\infty} \left[\left(\sum_{k=1}^{n} \frac{1}{k} \right) - \log(n) \right]$$

(which converges notoriously slowly):

```
>>> f = lambda n: sum([mpf(1)/k for k in range(1,int(n)+1)]) - log(n)
>>> limit(f, inf)
0.577215664901532860606512090082
>>> +euler
0.577215664901532860606512090082
```

With default settings, the following limit converges too slowly to be evaluated accurately. Changing to exponential sampling however gives a perfect result:

```
>>> f = lambda x: sqrt(x**3+x**2)/(sqrt(x**3)+x)
>>> limit(f, inf)
0.99283115855833028112924968649 1
>>> limit(f, inf, exp=True)
1.0
```

Extrapolation The following functions provide a direct interface to extrapolation algorithms. `nsum()` and `limit()` essentially work by calling the following functions with an increasing number of terms until the extrapolated limit is accurate enough.

The following functions may be useful to call directly if the precise number of terms needed to achieve a desired accuracy is known in advance, or if one wishes to study the convergence properties of the algorithms.

richardson()

mpmath.**richardson**(*ctx*, *seq*)

Given a list `seq` of the first N elements of a slowly convergent infinite sequence, `richardson()` (page 893) computes the N-term Richardson extrapolate for the limit.

`richardson()` (page 893) returns (v, c) where v is the estimated limit and c is the magnitude of the largest weight used during the computation. The weight provides an estimate of the precision lost to cancellation. Due to cancellation effects, the sequence must be typically be computed at a much higher precision than the target accuracy of the extrapolation.

Applicability and issues

The N-step Richardson extrapolation algorithm used by `richardson()` (page 893) is described in [1].

Richardson extrapolation only works for a specific type of sequence, namely one converging like partial sums of $P(1)/Q(1) + P(2)/Q(2) + \ldots$ where P and Q are polynomials. When the sequence does not convergence at such a rate `richardson()` (page 893) generally produces garbage.

Richardson extrapolation has the advantage of being fast: the N-term extrapolate requires only $O(N)$ arithmetic operations, and usually produces an estimate that is accurate to $O(N)$ digits. Contrast with the Shanks transformation (see `shanks()` (page 894)), which requires $O(N^2)$ operations.

`richardson()` (page 893) is unable to produce an estimate for the approximation error. One way to estimate the error is to perform two extrapolations with slightly different N and comparing the results.

Richardson extrapolation does not work for oscillating sequences. As a simple workaround, `richardson()` (page 893) detects if the last three elements do not differ monotonically, and in that case applies extrapolation only to the even-index elements.

Examples

Applying Richardson extrapolation to the Leibniz series for π:

```
>>> from mpmath import *
>>> mp.dps = 30; mp.pretty = True
>>> S = [4*sum(mpf(-1)**n/(2*n+1) for n in range(m))
...     for m in range(1,30)]
```

```
>>> v, c = richardson(S[:10])
>>> v
3.2126984126984126984126984127
>>> nprint([v-pi, c])
[0.0711058, 2.0]

>>> v, c = richardson(S[:30])
>>> v
3.1415926546862405282995420622 6
>>> nprint([v-pi, c])
[1.09645e-9, 20833.3]
```

References

1. [BenderOrszag] (page 1625) pp. 375-376

shanks()

mpmath.**shanks**(*ctx*, *seq*, *table=None*, *randomized=False*)

Given a list `seq` of the first N elements of a slowly convergent infinite sequence (A_k), `shanks()` (page 894) computes the iterated Shanks transformation $S(A), S(S(A)), \ldots, S^{N/2}(A)$. The Shanks transformation often provides strong convergence acceleration, especially if the sequence is oscillating.

The iterated Shanks transformation is computed using the Wynn epsilon algorithm (see [1]). `shanks()` (page 894) returns the full epsilon table generated by Wynn's algorithm, which can be read off as follows:

- The table is a list of lists forming a lower triangular matrix, where higher row and column indices correspond to more accurate values.
- The columns with even index hold dummy entries (required for the computation) and the columns with odd index hold the actual extrapolates.
- The last element in the last row is typically the most accurate estimate of the limit.
- The difference to the third last element in the last row provides an estimate of the approximation error.
- The magnitude of the second last element provides an estimate of the numerical accuracy lost to cancellation.

For convenience, so the extrapolation is stopped at an odd index so that `shanks(seq)[-1][-1]` always gives an estimate of the limit.

Optionally, an existing table can be passed to `shanks()` (page 894). This can be used to efficiently extend a previous computation after new elements have been appended to the sequence. The table will then be updated in-place.

The Shanks transformation

The Shanks transformation is defined as follows (see [2]): given the input sequence $(A_0, A_1, \ldots)$, the transformed sequence is given by

$$S(A_k) = \frac{A_{k+1}A_{k-1} - A_k^2}{A_{k+1} + A_{k-1} - 2A_k}$$

The Shanks transformation gives the exact limit A_∞ in a single step if $A_k = A + aq^k$. Note in particular that it extrapolates the exact sum of a geometric series in a single step.

Applying the Shanks transformation once often improves convergence substantially for an arbitrary sequence, but the optimal effect is obtained by applying it iteratively: $S(S(A_k)), S(S(S(A_k))), \ldots$.

Wynn's epsilon algorithm provides an efficient way to generate the table of iterated Shanks transformations. It reduces the computation of each element to essentially a single division, at the cost of requiring dummy elements in the table. See [1] for details.

Precision issues

Due to cancellation effects, the sequence must be typically be computed at a much higher precision than the target accuracy of the extrapolation.

If the Shanks transformation converges to the exact limit (such as if the sequence is a geometric series), then a division by zero occurs. By default, `shanks()` (page 894) handles this case by terminating the iteration and returning the table it has generated so far. With *randomized=True*, it will instead replace the zero by a pseudorandom number close to zero. (TODO: find a better solution to this problem.)

Examples

We illustrate by applying Shanks transformation to the Leibniz series for π:

```
>>> from mpmath import *
>>> mp.dps = 50
>>> S = [4*sum(mpf(-1)**n/(2*n+1) for n in range(m))
...     for m in range(1,30)]
>>>
>>> T = shanks(S[:7])
>>> for row in T:
...     nprint(row)
...
[-0.75]
[1.25, 3.16667]
[-1.75, 3.13333, -28.75]
[2.25, 3.14524, 82.25, 3.14234]
[-2.75, 3.13968, -177.75, 3.14139, -969.937]
[3.25, 3.14271, 327.25, 3.14166, 3515.06, 3.14161]
```

The extrapolated accuracy is about 4 digits, and about 4 digits may have been lost due to cancellation:

```
>>> L = T[-1]
>>> nprint([abs(L[-1] - pi), abs(L[-1] - L[-3]), abs(L[-2])])
[2.22532e-5, 4.78309e-5, 3515.06]
```

Now we extend the computation:

```
>>> T = shanks(S[:25], T)
>>> L = T[-1]
>>> nprint([abs(L[-1] - pi), abs(L[-1] - L[-3]), abs(L[-2])])
[3.75527e-19, 1.48478e-19, 2.96014e+17]
```

The value for pi is now accurate to 18 digits. About 18 digits may also have been lost to cancellation.

Here is an example with a geometric series, where the convergence is immediate (the sum is exactly 1):

```
>>> mp.dps = 15
>>> for row in shanks([0.5, 0.75, 0.875, 0.9375, 0.96875]):
...     nprint(row)
[4.0]
[8.0, 1.0]
```

References

1. [GravesMorris] (page 1625)
2. [BenderOrszag] (page 1625) pp. 368-375

Differentiation

Numerical derivatives (diff, diffs)

mpmath.**diff**(*ctx, f, x, n=1, **options*)

Numerically computes the derivative of f, $f'(x)$, or generally for an integer $n \geq 0$, the n-th derivative $f^{(n)}(x)$. A few basic examples are:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> diff(lambda x: x**2 + x, 1.0)
3.0
>>> diff(lambda x: x**2 + x, 1.0, 2)
2.0
>>> diff(lambda x: x**2 + x, 1.0, 3)
0.0
>>> nprint([diff(exp, 3, n) for n in range(5)])   # exp'(x) = exp(x)
[20.0855, 20.0855, 20.0855, 20.0855, 20.0855]
```

Even more generally, given a tuple of arguments $(x_1, \ldots, x_k)$ and order $(n_1, \ldots, n_k)$, the partial derivative $f^{(n_1,\ldots,n_k)}(x_1, \ldots, x_k)$ is evaluated. For example:

```
>>> diff(lambda x,y: 3*x*y + 2*y - x, (0.25, 0.5), (0,1))
2.75
>>> diff(lambda x,y: 3*x*y + 2*y - x, (0.25, 0.5), (1,1))
3.0
```

Options

The following optional keyword arguments are recognized:

method Supported methods are `'step'` or `'quad'`: derivatives may be computed using either a finite difference with a small step size h (default), or numerical quadrature.

direction Direction of finite difference: can be -1 for a left difference, 0 for a central difference (default), or +1 for a right difference; more generally can be any complex number.

addprec Extra precision for h used to account for the function's sensitivity to perturbations (default = 10).

relative Choose h relative to the magnitude of x, rather than an absolute value; useful for large or tiny x (default = False).

h As an alternative to `addprec` and `relative`, manually select the step size h.

singular If True, evaluation exactly at the point x is avoided; this is useful for differentiating functions with removable singularities. Default = False.

radius Radius of integration contour (with `method = 'quad'`). Default = 0.25. A larger radius typically is faster and more accurate, but it must be chosen so that f has no singularities within the radius from the evaluation point.

A finite difference requires $n+1$ function evaluations and must be performed at $(n+1)$ times the target precision. Accordingly, f must support fast evaluation at high precision.

With integration, a larger number of function evaluations is required, but not much extra precision is required. For high order derivatives, this method may thus be faster if f is very expensive to evaluate at high precision.

Further examples

The direction option is useful for computing left- or right-sided derivatives of nonsmooth functions:

```
>>> diff(abs, 0, direction=0)
0.0
>>> diff(abs, 0, direction=1)
1.0
>>> diff(abs, 0, direction=-1)
-1.0
```

More generally, if the direction is nonzero, a right difference is computed where the step size is multiplied by sign(direction). For example, with direction=+j, the derivative from the positive imaginary direction will be computed:

```
>>> diff(abs, 0, direction=j)
(0.0 - 1.0j)
```

With integration, the result may have a small imaginary part even even if the result is purely real:

```
>>> diff(sqrt, 1, method='quad')
(0.5 - 4.59...e-26j)
>>> chop(_)
0.5
```

Adding precision to obtain an accurate value:

```
>>> diff(cos, 1e-30)
0.0
>>> diff(cos, 1e-30, h=0.0001)
-9.99999998328279e-31
>>> diff(cos, 1e-30, addprec=100)
-1.0e-30
```

mpmath.**diffs**(*ctx*, *f*, *x*, *n=None*, ***options*)

Returns a generator that yields the sequence of derivatives

$$f(x), f'(x), f''(x), \ldots, f^{(k)}(x), \ldots$$

With `method='step'`, `diffs()` (page 897) uses only $O(k)$ function evaluations to generate the first k derivatives, rather than the roughly $O(k^2)$ evaluations required if one calls `diff()` (page 896) k separate times.

With $n < \infty$, the generator stops as soon as the n-th derivative has been generated. If the exact number of needed derivatives is known in advance, this is further slightly more efficient.

Options are the same as for `diff()` (page 896).

Examples

```
>>> from mpmath import *
>>> mp.dps = 15
>>> nprint(list(diffs(cos, 1, 5)))
[0.540302, -0.841471, -0.540302, 0.841471, 0.540302, -0.841471]
>>> for i, d in zip(range(6), diffs(cos, 1)):
...     print("%s %s" % (i, d))
...
0 0.54030230586814
1 -0.841470984807897
```

```
2 -0.54030230586814
3 0.841470984807897
4 0.54030230586814
5 -0.841470984807897
```

Composition of derivatives (diffs_prod, diffs_exp)

mpmath.**diffs_prod**(*ctx, factors*)

Given a list of N iterables or generators yielding $f_k(x), f_k'(x), f_k''(x), \ldots$ for $k = 1, \ldots, N$, generate $g(x), g'(x), g''(x), \ldots$ where $g(x) = f_1(x) f_2(x) \cdots f_N(x)$.

At high precision and for large orders, this is typically more efficient than numerical differentiation if the derivatives of each $f_k(x)$ admit direct computation.

Note: This function does not increase the working precision internally, so guard digits may have to be added externally for full accuracy.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = lambda x: exp(x)*cos(x)*sin(x)
>>> u = diffs(f, 1)
>>> v = mp.diffs_prod([diffs(exp,1), diffs(cos,1), diffs(sin,1)])
>>> next(u); next(v)
1.23586333600241
1.23586333600241
>>> next(u); next(v)
0.104658952245596
0.104658952245596
>>> next(u); next(v)
-5.96999877552086
-5.96999877552086
>>> next(u); next(v)
-12.4632923122697
-12.4632923122697
```

mpmath.**diffs_exp**(*ctx, fdiffs*)

Given an iterable or generator yielding $f(x), f'(x), f''(x), \ldots$ generate $g(x), g'(x), g''(x), \ldots$ where $g(x) = \exp(f(x))$.

At high precision and for large orders, this is typically more efficient than numerical differentiation if the derivatives of $f(x)$ admit direct computation.

Note: This function does not increase the working precision internally, so guard digits may have to be added externally for full accuracy.

Examples

The derivatives of the gamma function can be computed using logarithmic differentiation:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>>
>>> def diffs_loggamma(x):
...     yield loggamma(x)
...     i = 0
...     while 1:
...         yield psi(i,x)
...         i += 1
```

```
...
>>> u = diffs_exp(diffs_loggamma(3))
>>> v = diffs(gamma, 3)
>>> next(u); next(v)
2.0
2.0
>>> next(u); next(v)
1.84556867019693
1.84556867019693
>>> next(u); next(v)
2.49292999190269
2.49292999190269
>>> next(u); next(v)
3.44996501352367
3.44996501352367
```

Fractional derivatives / differintegration (differint)

mpmath.**differint**(*ctx, f, x, n=1, x0=0*)

Calculates the Riemann-Liouville differintegral, or fractional derivative, defined by

$$ {}_{x_0}\mathbb{D}_x^n f(x) \frac{1}{\Gamma(m-n)} \frac{d^m}{dx^m} \int_{x_0}^{x} (x-t)^{m-n-1} f(t) dt $$

where f is a given (presumably well-behaved) function, x is the evaluation point, n is the order, and x_0 is the reference point of integration (m is an arbitrary parameter selected automatically).

With $n = 1$, this is just the standard derivative $f'(x)$; with $n = 2$, the second derivative $f''(x)$, etc. With $n = -1$, it gives $\int_{x_0}^{x} f(t)dt$, with $n = -2$ it gives $\int_{x_0}^{x} \left(\int_{x_0}^{t} f(u)du\right) dt$, etc.

As n is permitted to be any number, this operator generalizes iterated differentiation and iterated integration to a single operator with a continuous order parameter.

Examples

There is an exact formula for the fractional derivative of a monomial x^p, which may be used as a reference. For example, the following gives a half-derivative (order 0.5):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> x = mpf(3); p = 2; n = 0.5
>>> differint(lambda t: t**p, x, n)
7.81764019044672
>>> gamma(p+1)/gamma(p-n+1) * x**(p-n)
7.81764019044672
```

Another useful test function is the exponential function, whose integration / differentiation formula easy generalizes to arbitrary order. Here we first compute a third derivative, and then a triply nested integral. (The reference point x_0 is set to $-\infty$ to avoid nonzero endpoint terms.):

```
>>> differint(lambda x: exp(pi*x), -1.5, 3)
0.278538406900792
>>> exp(pi*-1.5) * pi**3
0.278538406900792
>>> differint(lambda x: exp(pi*x), 3.5, -3, -inf)
1922.50563031149
>>> exp(pi*3.5) / pi**3
1922.50563031149
```

However, for noninteger n, the differentiation formula for the exponential function must be modified to give the same result as the Riemann-Liouville differintegral:

```
>>> x = mpf(3.5)
>>> c = pi
>>> n = 1+2*j
>>> differint(lambda x: exp(c*x), x, n)
(-123295.005390743 + 140955.117867654j)
>>> x**(-n) * exp(c)**x * (x*c)**n * gammainc(-n, 0, x*c) / gamma(-n)
(-123295.005390743 + 140955.117867654j)
```

Numerical integration (quadrature)

Standard quadrature (quad)

`mpmath.quad(ctx, f, *points, **kwargs)`

Computes a single, double or triple integral over a given 1D interval, 2D rectangle, or 3D cuboid. A basic example:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> quad(sin, [0, pi])
2.0
```

A basic 2D integral:

```
>>> f = lambda x, y: cos(x+y/2)
>>> quad(f, [-pi/2, pi/2], [0, pi])
4.0
```

Interval format

The integration range for each dimension may be specified using a list or tuple. Arguments are interpreted as follows:

`quad(f, [x1, x2])` – calculates $\int_{x_1}^{x_2} f(x)\,dx$

`quad(f, [x1, x2], [y1, y2])` – calculates $\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x,y)\,dy\,dx$

`quad(f, [x1, x2], [y1, y2], [z1, z2])` – calculates $\int_{x_1}^{x_2} \int_{y_1}^{y_2} \int_{z_1}^{z_2} f(x,y,z)\,dz\,dy\,dx$

Endpoints may be finite or infinite. An interval descriptor may also contain more than two points. In this case, the integration is split into subintervals, between each pair of consecutive points. This is useful for dealing with mid-interval discontinuities, or integrating over large intervals where the function is irregular or oscillates.

Options

`quad()` (page 900) recognizes the following keyword arguments:

method Chooses integration algorithm (described below).

error If set to true, `quad()` (page 900) returns (v, e) where v is the integral and e is the estimated error.

maxdegree Maximum degree of the quadrature rule to try before quitting.

verbose Print details about progress.

Algorithms

Mpmath presently implements two integration algorithms: tanh-sinh quadrature and Gauss-Legendre quadrature. These can be selected using *method='tanh-sinh'* or *method='gauss-legendre'* or by passing the classes *method=TanhSinh*, *method=GaussLegendre*. The functions `quadts()` and `quadgl()` are also available as shortcuts.

Both algorithms have the property that doubling the number of evaluation points roughly doubles the accuracy, so both are ideal for high precision quadrature (hundreds or thousands of digits).

At high precision, computing the nodes and weights for the integration can be expensive (more expensive than computing the function values). To make repeated integrations fast, nodes are automatically cached.

The advantages of the tanh-sinh algorithm are that it tends to handle endpoint singularities well, and that the nodes are cheap to compute on the first run. For these reasons, it is used by `quad()` (page 900) as the default algorithm.

Gauss-Legendre quadrature often requires fewer function evaluations, and is therefore often faster for repeated use, but the algorithm does not handle endpoint singularities as well and the nodes are more expensive to compute. Gauss-Legendre quadrature can be a better choice if the integrand is smooth and repeated integrations are required (e.g. for multiple integrals).

See the documentation for `TanhSinh` and `GaussLegendre` for additional details.

Examples of 1D integrals

Intervals may be infinite or half-infinite. The following two examples evaluate the limits of the inverse tangent function ($\int 1/(1+x^2) = \tan^{-1} x$), and the Gaussian integral $\int_{\infty}^{\infty} \exp(-x^2)\,dx = \sqrt{\pi}$:

```
>>> mp.dps = 15
>>> quad(lambda x: 2/(x**2+1), [0, inf])
3.14159265358979
>>> quad(lambda x: exp(-x**2), [-inf, inf])**2
3.14159265358979
```

Integrals can typically be resolved to high precision. The following computes 50 digits of π by integrating the area of the half-circle defined by $x^2 + y^2 \le 1$, $-1 \le x \le 1$, $y \ge 0$:

```
>>> mp.dps = 50
>>> 2*quad(lambda x: sqrt(1-x**2), [-1, 1])
3.1415926535897932384626433832795028841971693993751
```

One can just as well compute 1000 digits (output truncated):

```
>>> mp.dps = 1000
>>> 2*quad(lambda x: sqrt(1-x**2), [-1, 1])
3.1415926535897932384626433832795028841971693993751
```

Complex integrals are supported. The following computes a residue at $z = 0$ by integrating counterclockwise along the diamond-shaped path from 1 to $+i$ to -1 to $-i$ to 1:

```
>>> mp.dps = 15
>>> chop(quad(lambda z: 1/z, [1,j,-1,-j,1]))
(0.0 + 6.28318530717959j)
```

Examples of 2D and 3D integrals

Here are several nice examples of analytically solvable 2D integrals (taken from MathWorld [1]) that can be evaluated to high precision fairly rapidly by `quad()` (page 900):

```
>>> mp.dps = 30
>>> f = lambda x, y: (x-1)/((1-x*y)*log(x*y))
>>> quad(f, [0, 1], [0, 1])
0.577215664901532860606512090082
>>> +euler
0.577215664901532860606512090082

>>> f = lambda x, y: 1/sqrt(1+x**2+y**2)
>>> quad(f, [-1, 1], [-1, 1])
3.173436485306071342191756467 05
>>> 4*log(2+sqrt(3))-2*pi/3
3.173436485306071342191756467 05

>>> f = lambda x, y: 1/(1-x**2 * y**2)
>>> quad(f, [0, 1], [0, 1])
1.233700550136169827354311374 98
>>> pi**2 / 8
1.233700550136169827354311374 98

>>> quad(lambda x, y: 1/(1-x*y), [0, 1], [0, 1])
1.644934066848226436472415166 65
>>> pi**2 / 6
1.644934066848226436472415166 65
```

Multiple integrals may be done over infinite ranges:

```
>>> mp.dps = 15
>>> print(quad(lambda x,y: exp(-x-y), [0, inf], [1, inf]))
0.367879441171442
>>> print(1/e)
0.367879441171442
```

For nonrectangular areas, one can call `quad()` (page 900) recursively. For example, we can replicate the earlier example of calculating π by integrating over the unit-circle, and actually use double quadrature to actually measure the area circle:

```
>>> f = lambda x: quad(lambda y: 1, [-sqrt(1-x**2), sqrt(1-x**2)])
>>> quad(f, [-1, 1])
3.14159265358979
```

Here is a simple triple integral:

```
>>> mp.dps = 15
>>> f = lambda x,y,z: x*y/(1+z)
>>> quad(f, [0,1], [0,1], [1,2], method='gauss-legendre')
0.101366277027041
>>> (log(3)-log(2))/4
0.101366277027041
```

Singularities

Both tanh-sinh and Gauss-Legendre quadrature are designed to integrate smooth (infinitely differentiable) functions. Neither algorithm copes well with mid-interval singularities (such as mid-interval discontinuities in $f(x)$ or $f'(x)$). The best solution is to split the integral into parts:

```
>>> mp.dps = 15
>>> quad(lambda x: abs(sin(x)), [0, 2*pi])   # Bad
3.99900894176779
```

```
>>> quad(lambda x: abs(sin(x)), [0, pi, 2*pi])  # Good
4.0
```

The tanh-sinh rule often works well for integrands having a singularity at one or both endpoints:

```
>>> mp.dps = 15
>>> quad(log, [0, 1], method='tanh-sinh')  # Good
-1.0
>>> quad(log, [0, 1], method='gauss-legendre')  # Bad
-0.999932197413801
```

However, the result may still be inaccurate for some functions:

```
>>> quad(lambda x: 1/sqrt(x), [0, 1], method='tanh-sinh')
1.99999999946942
```

This problem is not due to the quadrature rule per se, but to numerical amplification of errors in the nodes. The problem can be circumvented by temporarily increasing the precision:

```
>>> mp.dps = 30
>>> a = quad(lambda x: 1/sqrt(x), [0, 1], method='tanh-sinh')
>>> mp.dps = 15
>>> +a
2.0
```

Highly variable functions

For functions that are smooth (in the sense of being infinitely differentiable) but contain sharp mid-interval peaks or many "bumps", `quad()` (page 900) may fail to provide full accuracy. For example, with default settings, `quad()` (page 900) is able to integrate $\sin(x)$ accurately over an interval of length 100 but not over length 1000:

```
>>> quad(sin, [0, 100]); 1-cos(100)   # Good
0.137681127712316
0.137681127712316
>>> quad(sin, [0, 1000]); 1-cos(1000)   # Bad
-37.8587612408485
0.437620923709297
```

One solution is to break the integration into 10 intervals of length 100:

```
>>> quad(sin, linspace(0, 1000, 10))   # Good
0.437620923709297
```

Another is to increase the degree of the quadrature:

```
>>> quad(sin, [0, 1000], maxdegree=10)   # Also good
0.437620923709297
```

Whether splitting the interval or increasing the degree is more efficient differs from case to case. Another example is the function $1/(1+x^2)$, which has a sharp peak centered around $x = 0$:

```
>>> f = lambda x: 1/(1+x**2)
>>> quad(f, [-100, 100])   # Bad
3.64804647105268
>>> quad(f, [-100, 100], maxdegree=10)   # Good
3.12159332021646
```

```
>>> quad(f, [-100, 0, 100])   # Also good
3.12159332021646
```

References

1. http://mathworld.wolfram.com/DoubleIntegral.html

Oscillatory quadrature (quadosc)

mpmath.**quadosc**(*ctx, f, interval, omega=None, period=None, zeros=None*)

Calculates

$$I = \int_a^b f(x)dx$$

where at least one of a and b is infinite and where $f(x) = g(x)\cos(\omega x + \phi)$ for some slowly decreasing function $g(x)$. With proper input, quadosc() (page 904) can also handle oscillatory integrals where the oscillation rate is different from a pure sine or cosine wave.

In the standard case when $|a| < \infty, b = \infty$, quadosc() (page 904) works by evaluating the infinite series

$$I = \int_a^{x_1} f(x)dx + \sum_{k=1}^{\infty} \int_{x_k}^{x_{k+1}} f(x)dx$$

where x_k are consecutive zeros (alternatively some other periodic reference point) of $f(x)$. Accordingly, quadosc() (page 904) requires information about the zeros of $f(x)$. For a periodic function, you can specify the zeros by either providing the angular frequency ω (*omega*) or the *period* $2\pi/\omega$. In general, you can specify the n-th zero by providing the *zeros* arguments. Below is an example of each:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = lambda x: sin(3*x)/(x**2+1)
>>> quadosc(f, [0,inf], omega=3)
0.37833007080198
>>> quadosc(f, [0,inf], period=2*pi/3)
0.37833007080198
>>> quadosc(f, [0,inf], zeros=lambda n: pi*n/3)
0.37833007080198
>>> (ei(3)*exp(-3)-exp(3)*ei(-3))/2  # Computed by Mathematica
0.37833007080198
```

Note that *zeros* was specified to multiply n by the *half-period*, not the full period. In theory, it does not matter whether each partial integral is done over a half period or a full period. However, if done over half-periods, the infinite series passed to nsum() (page 882) becomes an *alternating series* and this typically makes the extrapolation much more efficient.

Here is an example of an integration over the entire real line, and a half-infinite integration starting at $-\infty$:

```
>>> quadosc(lambda x: cos(x)/(1+x**2), [-inf, inf], omega=1)
1.15572734979092
>>> pi/e
1.15572734979092
>>> quadosc(lambda x: cos(x)/x**2, [-inf, -1], period=2*pi)
-0.0844109505595739
>>> cos(1)+si(1)-pi/2
-0.0844109505595738
```

Of course, the integrand may contain a complex exponential just as well as a real sine or cosine:

```
>>> quadosc(lambda x: exp(3*j*x)/(1+x**2), [-inf,inf], omega=3)
(0.156410688228254 + 0.0j)
>>> pi/e**3
0.156410688228254
>>> quadosc(lambda x: exp(3*j*x)/(2+x+x**2), [-inf,inf], omega=3)
(0.00317486988463794 - 0.0447701735209082j)
>>> 2*pi/sqrt(7)/exp(3*(j+sqrt(7))/2)
(0.00317486988463794 - 0.0447701735209082j)
```

Non-periodic functions

If $f(x) = g(x)h(x)$ for some function $h(x)$ that is not strictly periodic, *omega* or *period* might not work, and it might be necessary to use *zeros*.

A notable exception can be made for Bessel functions which, though not periodic, are "asymptotically periodic" in a sufficiently strong sense that the sum extrapolation will work out:

```
>>> quadosc(j0, [0, inf], period=2*pi)
1.0
>>> quadosc(j1, [0, inf], period=2*pi)
1.0
```

More properly, one should provide the exact Bessel function zeros:

```
>>> j0zero = lambda n: findroot(j0, pi*(n-0.25))
>>> quadosc(j0, [0, inf], zeros=j0zero)
1.0
```

For an example where *zeros* becomes necessary, consider the complete Fresnel integrals

$$\int_0^\infty \cos x^2\,dx = \int_0^\infty \sin x^2\,dx = \sqrt{\frac{\pi}{8}}.$$

Although the integrands do not decrease in magnitude as $x \to \infty$, the integrals are convergent since the oscillation rate increases (causing consecutive periods to asymptotically cancel out). These integrals are virtually impossible to calculate to any kind of accuracy using standard quadrature rules. However, if one provides the correct asymptotic distribution of zeros ($x_n \sim \sqrt{n}$), `quadosc()` (page 904) works:

```
>>> mp.dps = 30
>>> f = lambda x: cos(x**2)
>>> quadosc(f, [0,inf], zeros=lambda n:sqrt(pi*n))
0.626657068657750125603941321203
>>> f = lambda x: sin(x**2)
>>> quadosc(f, [0,inf], zeros=lambda n:sqrt(pi*n))
0.626657068657750125603941321203
>>> sqrt(pi/8)
0.626657068657750125603941321203
```

(Interestingly, these integrals can still be evaluated if one places some other constant than π in the square root sign.)

In general, if $f(x) \sim g(x)\cos(h(x))$, the zeros follow the inverse-function distribution $h^{-1}(x)$:

```
>>> mp.dps = 15
>>> f = lambda x: sin(exp(x))
>>> quadosc(f, [1,inf], zeros=lambda n: log(n))
-0.25024394235267
>>> pi/2-si(e)
-0.250243942352671
```

Non-alternating functions

If the integrand oscillates around a positive value, without alternating signs, the extrapolation might fail. A simple trick that sometimes works is to multiply or divide the frequency by 2:

```
>>> f = lambda x: 1/x**2+sin(x)/x**4
>>> quadosc(f, [1,inf], omega=1)  # Bad
1.28642190869861
>>> quadosc(f, [1,inf], omega=0.5)  # Perfect
1.28652953559617
>>> 1+(cos(1)+ci(1)+sin(1))/6
1.28652953559617
```

Fast decay

`quadosc()` (page 904) is primarily useful for slowly decaying integrands. If the integrand decreases exponentially or faster, `quad()` (page 900) will likely handle it without trouble (and generally be much faster than `quadosc()` (page 904)):

```
>>> quadosc(lambda x: cos(x)/exp(x), [0, inf], omega=1)
0.5
>>> quad(lambda x: cos(x)/exp(x), [0, inf])
0.5
```

Quadrature rules

class `mpmath.calculus.quadrature.`**`QuadratureRule`**(*ctx*)

Quadrature rules are implemented using this class, in order to simplify the code and provide a common infrastructure for tasks such as error estimation and node caching.

You can implement a custom quadrature rule by subclassing `QuadratureRule` and implementing the appropriate methods. The subclass can then be used by `quad()` (page 900) by passing it as the *method* argument.

`QuadratureRule` instances are supposed to be singletons. `QuadratureRule` therefore implements instance caching in `__new__()`.

`calc_nodes`(*degree*, *prec*, *verbose=False*)

Compute nodes for the standard interval $[-1, 1]$. Subclasses should probably implement only this method, and use `get_nodes()` method to retrieve the nodes.

`clear`()

Delete cached node data.

`estimate_error`(*results*, *prec*, *epsilon*)

Given results from integrations $[I_1, I_2, \ldots, I_k]$ done with a quadrature of rule of degree $1, 2, \ldots, k$, estimate the error of I_k.

For $k = 2$, we estimate $|I_\infty - I_2|$ as $|I_2 - I_1|$.

For $k > 2$, we extrapolate $|I_\infty - I_k| \approx |I_{k+1} - I_k|$ from $|I_k - I_{k-1}|$ and $|I_k - I_{k-2}|$ under the assumption that each degree increment roughly doubles the accuracy of

the quadrature rule (this is true for both `TanhSinh` and `GaussLegendre`). The extrapolation formula is given by Borwein, Bailey & Girgensohn. Although not very conservative, this method seems to be very robust in practice.

get_nodes(*a*, *b*, *degree*, *prec*, *verbose=False*)
Return nodes for given interval, degree and precision. The nodes are retrieved from a cache if already computed; otherwise they are computed by calling `calc_nodes()` and are then cached.

Subclasses should probably not implement this method, but just implement `calc_nodes()` for the actual node computation.

guess_degree(*prec*)
Given a desired precision p in bits, estimate the degree m of the quadrature required to accomplish full accuracy for typical integrals. By default, `quad()` (page 900) will perform up to m iterations. The value of m should be a slight overestimate, so that "slightly bad" integrals can be dealt with automatically using a few extra iterations. On the other hand, it should not be too big, so `quad()` (page 900) can quit within a reasonable amount of time when it is given an "unsolvable" integral.

The default formula used by `guess_degree()` is tuned for both `TanhSinh` and `GaussLegendre`. The output is roughly as follows:

p	m
50	6
100	7
500	10
3000	12

This formula is based purely on a limited amount of experimentation and will sometimes be wrong.

sum_next(*f*, *nodes*, *degree*, *prec*, *previous*, *verbose=False*)
Evaluates the step sum $\sum w_k f(x_k)$ where the *nodes* list contains the (w_k, x_k) pairs.

`summation()` will supply the list *results* of values computed by `sum_next()` at previous degrees, in case the quadrature rule is able to reuse them.

summation(*f*, *points*, *prec*, *epsilon*, *max_degree*, *verbose=False*)
Main integration function. Computes the 1D integral over the interval specified by *points*. For each subinterval, performs quadrature of degree from 1 up to *max_degree* until `estimate_error()` signals convergence.

`summation()` transforms each subintegration to the standard interval and then calls `sum_next()`.

transform_nodes(*nodes*, *a*, *b*, *verbose=False*)
Rescale standardized nodes (for $[-1, 1]$) to a general interval $[a, b]$. For a finite interval, a simple linear change of variables is used. Otherwise, the following transformations are used:

$$[a, \infty] : t = \frac{1}{x} + (a - 1)$$
$$[-\infty, b] : t = (b + 1) - \frac{1}{x}$$
$$[-\infty, \infty] : t = \frac{x}{\sqrt{1 - x^2}}$$

Tanh-sinh rule

class mpmath.calculus.quadrature.**TanhSinh**(*ctx*)

This class implements "tanh-sinh" or "doubly exponential" quadrature. This quadrature rule is based on the Euler-Maclaurin integral formula. By performing a change of variables involving nested exponentials / hyperbolic functions (hence the name), the derivatives at the endpoints vanish rapidly. Since the error term in the Euler-Maclaurin formula depends on the derivatives at the endpoints, a simple step sum becomes extremely accurate. In practice, this means that doubling the number of evaluation points roughly doubles the number of accurate digits.

Comparison to Gauss-Legendre:

- Initial computation of nodes is usually faster
- Handles endpoint singularities better
- Handles infinite integration intervals better
- Is slower for smooth integrands once nodes have been computed

The implementation of the tanh-sinh algorithm is based on the description given in Borwein, Bailey & Girgensohn, "Experimentation in Mathematics - Computational Paths to Discovery", A K Peters, 2003, pages 312-313. In the present implementation, a few improvements have been made:

- A more efficient scheme is used to compute nodes (exploiting recurrence for the exponential function)
- The nodes are computed successively instead of all at once

Various documents describing the algorithm are available online, e.g.:

- http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-tanh-sinh.pdf
- http://users.cs.dal.ca/~jborwein/tanh-sinh.pdf

calc_nodes(*degree*, *prec*, *verbose=False*)

The abscissas and weights for tanh-sinh quadrature of degree m are given by

$$x_k = \tanh(\pi/2 \sinh(t_k))$$
$$w_k = \pi/2 \cosh(t_k)/\cosh(\pi/2 \sinh(t_k))^2$$

where $t_k = t_0 + hk$ for a step length $h \sim 2^{-m}$. The list of nodes is actually infinite, but the weights die off so rapidly that only a few are needed.

sum_next(*f*, *nodes*, *degree*, *prec*, *previous*, *verbose=False*)

Step sum for tanh-sinh quadrature of degree m. We exploit the fact that half of the abscissas at degree m are precisely the abscissas from degree $m-1$. Thus reusing the result from the previous level allows a 2x speedup.

Gauss-Legendre rule

class mpmath.calculus.quadrature.**GaussLegendre**(*ctx*)

This class implements Gauss-Legendre quadrature, which is exceptionally efficient for polynomials and polynomial-like (i.e. very smooth) integrands.

The abscissas and weights are given by roots and values of Legendre polynomials, which are the orthogonal polynomials on $[-1, 1]$ with respect to the unit weight (see `legendre()` (page 774)).

In this implementation, we take the "degree" m of the quadrature to denote a Gauss-Legendre rule of degree $3 \cdot 2^m$ (following Borwein, Bailey & Girgensohn). This way we get quadratic, rather than linear, convergence as the degree is incremented.

Comparison to tanh-sinh quadrature:

- Is faster for smooth integrands once nodes have been computed
- Initial computation of nodes is usually slower
- Handles endpoint singularities worse
- Handles infinite integration intervals worse

calc_nodes(*degree*, *prec*, *verbose=False*)

Calculates the abscissas and weights for Gauss-Legendre quadrature of degree of given degree (actually $3 \cdot 2^m$).

Ordinary differential equations

Solving the ODE initial value problem (odefun)

mpmath.**odefun**(*ctx*, *F*, *x0*, *y0*, *tol=None*, *degree=None*, *method='taylor'*, *verbose=False*)

Returns a function $y(x) = [y_0(x), y_1(x), \ldots, y_n(x)]$ that is a numerical solution of the $n+1$-dimensional first-order ordinary differential equation (ODE) system

$$\begin{aligned} y_0'(x) &= F_0(x, [y_0(x), y_1(x), \ldots, y_n(x)]) \\ y_1'(x) &= F_1(x, [y_0(x), y_1(x), \ldots, y_n(x)]) \\ &\vdots \\ y_n'(x) &= F_n(x, [y_0(x), y_1(x), \ldots, y_n(x)]) \end{aligned}$$

The derivatives are specified by the vector-valued function *F* that evaluates $[y_0', \ldots, y_n'] = F(x, [y_0, \ldots, y_n])$. The initial point x_0 is specified by the scalar argument *x0*, and the initial value $y(x_0) = [y_0(x_0), \ldots, y_n(x_0)]$ is specified by the vector argument *y0*.

For convenience, if the system is one-dimensional, you may optionally provide just a scalar value for *y0*. In this case, *F* should accept a scalar *y* argument and return a scalar. The solution function *y* will return scalar values instead of length-1 vectors.

Evaluation of the solution function $y(x)$ is permitted for any $x \geq x_0$.

A high-order ODE can be solved by transforming it into first-order vector form. This transformation is described in standard texts on ODEs. Examples will also be given below.

Options, speed and accuracy

By default, `odefun()` (page 909) uses a high-order Taylor series method. For reasonably well-behaved problems, the solution will be fully accurate to within the working precision. Note that *F* must be possible to evaluate to very high precision for the generation of Taylor series to work.

To get a faster but less accurate solution, you can set a large value for *tol* (which defaults roughly to *eps*). If you just want to plot the solution or perform a basic simulation, *tol = 0.01* is likely sufficient.

The *degree* argument controls the degree of the solver (with *method='taylor'*, this is the degree of the Taylor series expansion). A higher degree means that a longer step can be taken before a new local solution must be generated from *F*, meaning that fewer steps are required to get from x_0 to a given x_1. On the other hand, a higher degree also means that each local solution becomes more expensive (i.e., more evaluations of *F* are required per step, and at higher precision).

The optimal setting therefore involves a tradeoff. Generally, decreasing the *degree* for Taylor series is likely to give faster solution at low precision, while increasing is likely to be better at higher precision.

The function object returned by odefun() (page 909) caches the solutions at all step points and uses polynomial interpolation between step points. Therefore, once $y(x_1)$ has been evaluated for some x_1, $y(x)$ can be evaluated very quickly for any $x_0 \le x \le x_1$. and continuing the evaluation up to $x_2 > x_1$ is also fast.

Examples of first-order ODEs

We will solve the standard test problem $y'(x) = y(x), y(0) = 1$ which has explicit solution $y(x) = \exp(x)$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = odefun(lambda x, y: y, 0, 1)
>>> for x in [0, 1, 2.5]:
...     print((f(x), exp(x)))
...
(1.0, 1.0)
(2.71828182845905, 2.71828182845905)
(12.1824939607035, 12.1824939607035)
```

The solution with high precision:

```
>>> mp.dps = 50
>>> f = odefun(lambda x, y: y, 0, 1)
>>> f(1)
2.7182818284590452353602874713526624977572470937
>>> exp(1)
2.7182818284590452353602874713526624977572470937
```

Using the more general vectorized form, the test problem can be input as (note that *f* returns a 1-element vector):

```
>>> mp.dps = 15
>>> f = odefun(lambda x, y: [y[0]], 0, [1])
>>> f(1)
[2.71828182845905]
```

odefun() (page 909) can solve nonlinear ODEs, which are generally impossible (and at best difficult) to solve analytically. As an example of a nonlinear ODE, we will solve $y'(x) = x\sin(y(x))$ for $y(0) = \pi/2$. An exact solution happens to be known for this problem, and is given by $y(x) = 2\tan^{-1}\left(\exp\left(x^2/2\right)\right)$:

```
>>> f = odefun(lambda x, y: x*sin(y), 0, pi/2)
>>> for x in [2, 5, 10]:
...     print((f(x), 2*atan(exp(mpf(x)**2/2))))
...
(2.87255666284091, 2.87255666284091)
(3.14158520028345, 3.14158520028345)
(3.14159265358979, 3.14159265358979)
```

If F is independent of y, an ODE can be solved using direct integration. We can therefore obtain a reference solution with quad() (page 900):

```
>>> f = lambda x: (1+x**2)/(1+x**3)
>>> g = odefun(lambda x, y: f(x), pi, 0)
>>> g(2*pi)
0.72128263801696
>>> quad(f, [pi, 2*pi])
0.72128263801696
```

Examples of second-order ODEs

We will solve the harmonic oscillator equation $y''(x)+y(x)=0$. To do this, we introduce the helper functions $y_0 = y, y_1 = y_0'$ whereby the original equation can be written as $y_1'+y_0'=0$. Put together, we get the first-order, two-dimensional vector ODE

$$\begin{cases} y_0' = y_1 \\ y_1' = -y_0 \end{cases}$$

To get a well-defined IVP, we need two initial values. With $y(0) = y_0(0) = 1$ and $-y'(0) = y_1(0) = 0$, the problem will of course be solved by $y(x) = y_0(x) = \cos(x)$ and $-y'(x) = y_1(x) = \sin(x)$. We check this:

```
>>> f = odefun(lambda x, y: [-y[1], y[0]], 0, [1, 0])
>>> for x in [0, 1, 2.5, 10]:
...     nprint(f(x), 15)
...     nprint([cos(x), sin(x)], 15)
...     print("---")
...
[1.0, 0.0]
[1.0, 0.0]
---
[0.54030230586814, 0.841470984807897]
[0.54030230586814, 0.841470984807897]
---
[-0.801143615546934, 0.598472144103957]
[-0.801143615546934, 0.598472144103957]
---
[-0.839071529076452, -0.54402111088937]
[-0.839071529076452, -0.54402111088937]
---
```

Note that we get both the sine and the cosine solutions simultaneously.

TODO

- Better automatic choice of degree and step size
- Make determination of Taylor series convergence radius more robust
- Allow solution for $x < x_0$
- Allow solution for complex x
- Test for difficult (ill-conditioned) problems
- Implement Runge-Kutta and other algorithms

Function approximation

Taylor series (taylor)

`mpmath.`**`taylor`**(*ctx, f, x, n, **options*)

Produces a degree-n Taylor polynomial around the point x of the given function f. The coefficients are returned as a list.

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(chop(taylor(sin, 0, 5)))
[0.0, 1.0, 0.0, -0.166667, 0.0, 0.00833333]
```

The coefficients are computed using high-order numerical differentiation. The function must be possible to evaluate to arbitrary precision. See `diff()` (page 896) for additional details and supported keyword options.

Note that to evaluate the Taylor polynomial as an approximation of f, e.g. with `polyval()` (page 874), the coefficients must be reversed, and the point of the Taylor expansion must be subtracted from the argument:

```
>>> p = taylor(exp, 2.0, 10)
>>> polyval(p[::-1], 2.5 - 2.0)
12.1824939606092
>>> exp(2.5)
12.1824939607035
```

Pade approximation (pade)

`mpmath.pade(ctx, a, L, M)`

Computes a Pade approximation of degree (L, M) to a function. Given at least $L + M + 1$ Taylor coefficients a approximating a function $A(x)$, `pade()` (page 912) returns coefficients of polynomials P, Q satisfying

$$P = \sum_{k=0}^{L} p_k x^k$$
$$Q = \sum_{k=0}^{M} q_k x^k$$
$$Q_0 = 1$$
$$A(x)Q(x) = P(x) + O(x^{L+M+1})$$

$P(x)/Q(x)$ can provide a good approximation to an analytic function beyond the radius of convergence of its Taylor series (example from G.A. Baker 'Essentials of Pade Approximants' Academic Press, Ch.1A):

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> one = mpf(1)
>>> def f(x):
...     return sqrt((one + 2*x)/(one + x))
...
>>> a = taylor(f, 0, 6)
>>> p, q = pade(a, 3, 3)
>>> x = 10
>>> polyval(p[::-1], x)/polyval(q[::-1], x)
1.38169105566806
>>> f(x)
1.38169855941551
```

Chebyshev approximation (chebyfit)

`mpmath.chebyfit(ctx, f, interval, N, error=False)`

Computes a polynomial of degree $N - 1$ that approximates the given function f on the interval $[a, b]$. With `error=True`, `chebyfit()` (page 912) also returns an accurate estimate of the maximum absolute error; that is, the maximum value of $|f(x) - P(x)|$ for $x \in [a, b]$.

`chebyfit()` (page 912) uses the Chebyshev approximation formula, which gives a nearly optimal solution: that is, the maximum error of the approximating polynomial is very close to the smallest possible for any polynomial of the same degree.

Chebyshev approximation is very useful if one needs repeated evaluation of an expensive function, such as function defined implicitly by an integral or a differential equation. (For

example, it could be used to turn a slow mpmath function into a fast machine-precision version of the same.)

Examples

Here we use chebyfit() (page 912) to generate a low-degree approximation of $f(x) = \cos(x)$, valid on the interval $[1, 2]$:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> poly, err = chebyfit(cos, [1, 2], 5, error=True)
>>> nprint(poly)
[0.00291682, 0.146166, -0.732491, 0.174141, 0.949553]
>>> nprint(err, 12)
1.61351758081e-5
```

The polynomial can be evaluated using `polyval`:

```
>>> nprint(polyval(poly, 1.6), 12)
-0.0291858904138
>>> nprint(cos(1.6), 12)
-0.0291995223013
```

Sampling the true error at 1000 points shows that the error estimate generated by `chebyfit` is remarkably good:

```
>>> error = lambda x: abs(cos(x) - polyval(poly, x))
>>> nprint(max([error(1+n/1000.) for n in range(1000)]), 12)
1.61349954245e-5
```

Choice of degree

The degree N can be set arbitrarily high, to obtain an arbitrarily good approximation. As a rule of thumb, an N-term Chebyshev approximation is good to $N/(b-a)$ decimal places on a unit interval (although this depends on how well-behaved f is). The cost grows accordingly: `chebyfit` evaluates the function $(N^2)/2$ times to compute the coefficients and an additional N times to estimate the error.

Possible issues

One should be careful to use a sufficiently high working precision both when calling `chebyfit` and when evaluating the resulting polynomial, as the polynomial is sometimes ill-conditioned. It is for example difficult to reach 15-digit accuracy when evaluating the polynomial using machine precision floats, no matter the theoretical accuracy of the polynomial. (The option to return the coefficients in Chebyshev form should be made available in the future.)

It is important to note the Chebyshev approximation works poorly if f is not smooth. A function containing singularities, rapid oscillation, etc can be approximated more effectively by multiplying it by a weight function that cancels out the nonsmooth features, or by dividing the interval into several segments.

Fourier series (`fourier`, `fourierval`)

`mpmath.`**`fourier`**(*ctx*, *f*, *interval*, *N*)

Computes the Fourier series of degree N of the given function on the interval $[a, b]$. More precisely, fourier() (page 913) returns two lists (c, s) of coefficients (the cosine series and sine series, respectively), such that

$$f(x) \sim \sum_{k=0}^{N} c_k \cos(km) + s_k \sin(km)$$

where $m = 2\pi/(b-a)$.

Note that many texts define the first coefficient as $2c_0$ instead of c_0. The easiest way to evaluate the computed series correctly is to pass it to `fourierval()` (page 914).

Examples

The function $f(x) = x$ has a simple Fourier series on the standard interval $[-\pi, \pi]$. The cosine coefficients are all zero (because the function has odd symmetry), and the sine coefficients are rational numbers:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> c, s = fourier(lambda x: x, [-pi, pi], 5)
>>> nprint(c)
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint(s)
[0.0, 2.0, -1.0, 0.666667, -0.5, 0.4]
```

This computes a Fourier series of a nonsymmetric function on a nonstandard interval:

```
>>> I = [-1, 1.5]
>>> f = lambda x: x**2 - 4*x + 1
>>> cs = fourier(f, I, 4)
>>> nprint(cs[0])
[0.583333, 1.12479, -1.27552, 0.904708, -0.441296]
>>> nprint(cs[1])
[0.0, -2.6255, 0.580905, 0.219974, -0.540057]
```

It is instructive to plot a function along with its truncated Fourier series:

```
>>> plot([f, lambda x: fourierval(cs, I, x)], I)
```

Fourier series generally converge slowly (and may not converge pointwise). For example, if $f(x) = \cosh(x)$, a 10-term Fourier series gives an L^2 error corresponding to 2-digit accuracy:

```
>>> I = [-1, 1]
>>> cs = fourier(cosh, I, 9)
>>> g = lambda x: (cosh(x) - fourierval(cs, I, x))**2
>>> nprint(sqrt(quad(g, I)))
0.00467963
```

`fourier()` (page 913) uses numerical quadrature. For nonsmooth functions, the accuracy (and speed) can be improved by including all singular points in the interval specification:

```
>>> nprint(fourier(abs, [-1, 1], 0), 10)
([0.5000441648], [0.0])
>>> nprint(fourier(abs, [-1, 0, 1], 0), 10)
([0.5], [0.0])
```

mpmath.**fourierval**(*ctx, series, interval, x*)

Evaluates a Fourier series (in the format computed by by `fourier()` (page 913) for the given interval) at the point x.

The series should be a pair (c, s) where c is the cosine series and s is the sine series. The two lists need not have the same length.

Matrices

Creating matrices

Basic methods Matrices in mpmath are implemented using dictionaries. Only non-zero values are stored, so it is cheap to represent sparse matrices.

The most basic way to create one is to use the `matrix` class directly. You can create an empty matrix specifying the dimensions:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> matrix(2)
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
>>> matrix(2, 3)
matrix(
[['0.0', '0.0', '0.0'],
 ['0.0', '0.0', '0.0']])
```

Calling `matrix` with one dimension will create a square matrix.

To access the dimensions of a matrix, use the `rows` or `cols` keyword:

```
>>> A = matrix(3, 2)
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0'],
 ['0.0', '0.0']])
>>> A.rows
3
>>> A.cols
2
```

You can also change the dimension of an existing matrix. This will set the new elements to 0. If the new dimension is smaller than before, the concerning elements are discarded:

```
>>> A.rows = 2
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
```

Internally `convert` is applied every time an element is set. This is done using the syntax A[row,column], counting from 0:

```
>>> A = matrix(2)
>>> A[1,1] = 1 + 1j
>>> print A
[0.0           0.0]
[0.0  (1.0 + 1.0j)]
```

A more comfortable way to create a matrix lets you use nested lists:

```
>>> matrix([[1, 2], [3, 4]])
matrix(
[['1.0', '2.0'],
 ['3.0', '4.0']])
```

Advanced methods Convenient functions are available for creating various standard matrices:

```
>>> zeros(2)
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
>>> ones(2)
matrix(
[['1.0', '1.0'],
 ['1.0', '1.0']])
>>> diag([1, 2, 3]) # diagonal matrix
matrix(
[['1.0', '0.0', '0.0'],
 ['0.0', '2.0', '0.0'],
 ['0.0', '0.0', '3.0']])
>>> eye(2) # identity matrix
matrix(
[['1.0', '0.0'],
 ['0.0', '1.0']])
```

You can even create random matrices:

```
>>> randmatrix(2)
matrix(
[['0.53491598236191806', '0.57195669543302752'],
 ['0.85589992269513615', '0.82444367501382143']])
```

Vectors Vectors may also be represented by the `matrix` class (with rows = 1 or cols = 1). For vectors there are some things which make life easier. A column vector can be created using a flat list, a row vectors using an almost flat nested list:

```
>>> matrix([1, 2, 3])
matrix(
[['1.0'],
 ['2.0'],
 ['3.0']])
>>> matrix([[1, 2, 3]])
matrix(
[['1.0', '2.0', '3.0']])
```

Optionally vectors can be accessed like lists, using only a single index:

```
>>> x = matrix([1, 2, 3])
>>> x[1]
mpf('2.0')
>>> x[1,0]
mpf('2.0')
```

Other Like you probably expected, matrices can be printed:

```
>>> print randmatrix(3)
[ 0.782963853573023  0.802057689719883  0.427895717335467]
[0.0541876859348597  0.708243266653103  0.615134039977379]
[ 0.856151514955773  0.544759264818486  0.686210904770947]
```

Use `nstr` or `nprint` to specify the number of digits to print:

```
>>> nprint(randmatrix(5), 3)
[2.07e-1  1.66e-1  5.06e-1  1.89e-1  8.29e-1]
[6.62e-1  6.55e-1  4.47e-1  4.82e-1  2.06e-2]
[4.33e-1  7.75e-1  6.93e-2  2.86e-1  5.71e-1]
[1.01e-1  2.53e-1  6.13e-1  3.32e-1  2.59e-1]
[1.56e-1  7.27e-2  6.05e-1  6.67e-2  2.79e-1]
```

As matrices are mutable, you will need to copy them sometimes:

```
>>> A = matrix(2)
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
>>> B = A.copy()
>>> B[0,0] = 1
>>> B
matrix(
[['1.0', '0.0'],
 ['0.0', '0.0']])
>>> A
matrix(
[['0.0', '0.0'],
 ['0.0', '0.0']])
```

Finally, it is possible to convert a matrix to a nested list. This is very useful, as most Python libraries involving matrices or arrays (namely NumPy or SymPy) support this format:

```
>>> B.tolist()
[[mpf('1.0'), mpf('0.0')], [mpf('0.0'), mpf('0.0')]]
```

Matrix operations

You can add and substract matrices of compatible dimensions:

```
>>> A = matrix([[1, 2], [3, 4]])
>>> B = matrix([[-2, 4], [5, 9]])
>>> A + B
matrix(
[['-1.0', '6.0'],
 ['8.0', '13.0']])
>>> A - B
matrix(
[['3.0', '-2.0'],
 ['-2.0', '-5.0']])
>>> A + ones(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...", line 238, in __add__
    raise ValueError('incompatible dimensions for addition')
ValueError: incompatible dimensions for addition
```

It is possible to multiply or add matrices and scalars. In the latter case the operation will be done element-wise:

```
>>> A * 2
matrix(
```

```
[['2.0', '4.0'],
 ['6.0', '8.0']])
>>> A / 4
matrix(
[['0.25', '0.5'],
 ['0.75', '1.0']])
>>> A - 1
matrix(
[['0.0', '1.0'],
 ['2.0', '3.0']])
```

Of course you can perform matrix multiplication, if the dimensions are compatible:

```
>>> A * B
matrix(
[['8.0', '22.0'],
 ['14.0', '48.0']])
>>> matrix([[1, 2, 3]]) * matrix([[-6], [7], [-2]])
matrix(
[['2.0']])
```

You can raise powers of square matrices:

```
>>> A**2
matrix(
[['7.0', '10.0'],
 ['15.0', '22.0']])
```

Negative powers will calculate the inverse:

```
>>> A**-1
matrix(
[['-2.0', '1.0'],
 ['1.5', '-0.5']])
>>> nprint(A * A**-1, 3)
[      1.0  1.08e-19]
[-2.17e-19       1.0]
```

Matrix transposition is straightforward:

```
>>> A = ones(2, 3)
>>> A
matrix(
[['1.0', '1.0', '1.0'],
 ['1.0', '1.0', '1.0']])
>>> A.T
matrix(
[['1.0', '1.0'],
 ['1.0', '1.0'],
 ['1.0', '1.0']])
```

Norms Sometimes you need to know how "large" a matrix or vector is. Due to their multi-dimensional nature it's not possible to compare them, but there are several functions to map a matrix or a vector to a positive real number, the so called norms.

`mpmath.norm`(*ctx*, *x*, *p=2*)

Gives the entrywise p-norm of an iterable *x*, i.e. the vector norm $\left(\sum_k |x_k|^p\right)^{1/p}$, for any given $1 \le p \le \infty$.

Special cases:

If x is not iterable, this just returns `absmax(x)`.

`p=1` gives the sum of absolute values.

`p=2` is the standard Euclidean vector norm.

`p=inf` gives the magnitude of the largest element.

For x a matrix, `p=2` is the Frobenius norm. For operator matrix norms, use `mnorm()` (page 919) instead.

You can use the string 'inf' as well as float('inf') or mpf('inf') to specify the infinity norm.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> x = matrix([-10, 2, 100])
>>> norm(x, 1)
mpf('112.0')
>>> norm(x, 2)
mpf('100.5186549850325')
>>> norm(x, inf)
mpf('100.0')
```

mpmath.**mnorm**(*ctx, A, p=1*)

Gives the matrix (operator) p-norm of A. Currently `p=1` and `p=inf` are supported:

`p=1` gives the 1-norm (maximal column sum)

`p=inf` gives the ∞-norm (maximal row sum). You can use the string 'inf' as well as float('inf') or mpf('inf')

`p=2` (not implemented) for a square matrix is the usual spectral matrix norm, i.e. the largest singular value.

`p='f'` (or 'F', 'fro', 'Frobenius, 'frobenius') gives the Frobenius norm, which is the elementwise 2-norm. The Frobenius norm is an approximation of the spectral norm and satisfies

$$\frac{1}{\sqrt{\mathrm{rank}(A)}}\|A\|_F \le \|A\|_2 \le \|A\|_F$$

The Frobenius norm lacks some mathematical properties that might be expected of a norm.

For general elementwise p-norms, use `norm()` (page 918) instead.

Examples

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = False
>>> A = matrix([[1, -1000], [100, 50]])
>>> mnorm(A, 1)
mpf('1050.0')
>>> mnorm(A, inf)
mpf('1001.0')
>>> mnorm(A, 'F')
mpf('1006.2310867787777')
```

Linear algebra

Decompositions

`mpmath.cholesky(ctx, A, tol=None)`

Cholesky decomposition of a symmetric positive-definite matrix A. Returns a lower triangular matrix L such that $A = L \times L^T$. More generally, for a complex Hermitian positive-definite matrix, a Cholesky decomposition satisfying $A = L \times L^H$ is returned.

The Cholesky decomposition can be used to solve linear equation systems twice as efficiently as LU decomposition, or to test whether A is positive-definite.

The optional parameter `tol` determines the tolerance for verifying positive-definiteness.

Examples

Cholesky decomposition of a positive-definite symmetric matrix:

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> A = eye(3) + hilbert(3)
>>> nprint(A)
[     2.0      0.5  0.333333]
[     0.5  1.33333      0.25]
[0.333333     0.25       1.2]
>>> L = cholesky(A)
>>> nprint(L)
[ 1.41421      0.0      0.0]
[0.353553  1.09924      0.0]
[0.235702  0.15162  1.05899]
>>> chop(A - L*L.T)
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
```

Cholesky decomposition of a Hermitian matrix:

```
>>> A = eye(3) + matrix([[0,0.25j,-0.5j],[-0.25j,0,0],[0.5j,0,0]])
>>> L = cholesky(A)
>>> nprint(L)
[          1.0                0.0                0.0]
[(0.0 - 0.25j)  (0.968246 + 0.0j)                0.0]
[ (0.0 + 0.5j)  (0.129099 + 0.0j)  (0.856349 + 0.0j)]
>>> chop(A - L*L.H)
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
```

Attempted Cholesky decomposition of a matrix that is not positive definite:

```
>>> A = -eye(3) + hilbert(3)
>>> L = cholesky(A)
Traceback (most recent call last):
  ...
ValueError: matrix is not positive-definite
```

References

1. [Wikipedia] (page 1626) http://en.wikipedia.org/wiki/Cholesky_decomposition

Linear equations Basic linear algebra is implemented; you can for example solve the linear equation system:

```
  x + 2*y = -10
3*x + 4*y =  10
```

using `lu_solve`:

```
>>> A = matrix([[1, 2], [3, 4]])
>>> b = matrix([-10, 10])
>>> x = lu_solve(A, b)
>>> x
matrix(
[['30.0'],
 ['-20.0']])
```

If you don't trust the result, use `residual` to calculate the residual ||A*x-b||:

```
>>> residual(A, x, b)
matrix(
[['3.46944695195361e-18'],
 ['3.46944695195361e-18']])
>>> str(eps)
'2.22044604925031e-16'
```

As you can see, the solution is quite accurate. The error is caused by the inaccuracy of the internal floating point arithmetic. Though, it's even smaller than the current machine epsilon, which basically means you can trust the result.

If you need more speed, use NumPy, or use `fp` instead `mp` matrices and methods:

```
>>> A = fp.matrix([[1, 2], [3, 4]])
>>> b = fp.matrix([-10, 10])
>>> fp.lu_solve(A, b)
matrix(
[['30.0'],
 ['-20.0']])
```

`lu_solve` accepts overdetermined systems. It is usually not possible to solve such systems, so the residual is minimized instead. Internally this is done using Cholesky decomposition to compute a least squares approximation. This means that that `lu_solve` will square the errors. If you can't afford this, use `qr_solve` instead. It is twice as slow but more accurate, and it calculates the residual automatically.

Matrix factorization The function `lu` computes an explicit LU factorization of a matrix:

```
>>> P, L, U = lu(matrix([[0,2,3],[4,5,6],[7,8,9]]))
>>> print P
[0.0  0.0  1.0]
[1.0  0.0  0.0]
[0.0  1.0  0.0]
>>> print L
[              1.0                0.0  0.0]
[              0.0                1.0  0.0]
[0.571428571428571  0.214285714285714  1.0]
>>> print U
[7.0  8.0                9.0]
[0.0  2.0                3.0]
[0.0  0.0  0.214285714285714]
```

```
>>> print P.T*L*U
[0.0  2.0  3.0]
[4.0  5.0  6.0]
[7.0  8.0  9.0]
```

Interval and double-precision matrices

The `iv.matrix` and `fp.matrix` classes convert inputs to intervals and Python floating-point numbers respectively.

Interval matrices can be used to perform linear algebra operations with rigorous error tracking:

```
>>> a = iv.matrix([['0.1','0.3','1.0'],
...                ['7.1','5.5','4.8'],
...                ['3.2','4.4','5.6']])
>>>
>>> b = iv.matrix(['4','0.6','0.5'])
>>> c = iv.lu_solve(a, b)
>>> print c
[  [5.2582327113062393041, 5.2582327113062749951]]
[[-13.155049396267856583, -13.155049396267821167]]
[  [7.4206915477497212555, 7.4206915477497310922]]
>>> print a*c
[  [3.9999999999999866773, 4.0000000000000133227]]
[[0.59999999999972430942, 0.60000000000027142733]]
[[0.49999999999982236432, 0.50000000000018474111]]
```

Matrix functions

`mpmath.`**`expm`**(*ctx*, *A*, *method='taylor'*)

Computes the matrix exponential of a square matrix A, which is defined by the power series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots$$

With method='taylor', the matrix exponential is computed using the Taylor series. With method='pade', Pade approximants are used instead.

Examples

Basic examples:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> expm(zeros(3))
[1.0  0.0  0.0]
[0.0  1.0  0.0]
[0.0  0.0  1.0]
>>> expm(eye(3))
[2.71828182845905               0.0               0.0]
[             0.0  2.71828182845905               0.0]
[             0.0               0.0  2.71828182845905]
>>> expm([[1,1,0],[1,0,1],[0,1,0]])
[ 3.86814500615414  2.26812870852145  0.841130841230196]
```

```
[ 2.26812870852145  2.44114713886289   1.42699786729125]
[0.841130841230196  1.42699786729125    1.6000162976327]
>>> expm([[1,1,0],[1,0,1],[0,1,0]], method='pade')
[ 3.86814500615414  2.26812870852145  0.841130841230196]
[ 2.26812870852145  2.44114713886289   1.42699786729125]
[0.841130841230196  1.42699786729125    1.6000162976327]
>>> expm([[1+j, 0], [1+j,1]])
[(1.46869393991589 + 2.28735528717884j)                        0.0]
[  (1.03776739863568 + 3.536943175722j)  (2.71828182845905 + 0.0j)]
```

Matrices with large entries are allowed:

```
>>> expm(matrix([[1,2],[2,3]])**25)
[5.65024064048415e+2050488462815550  9.14228140091932e+2050488462815550]
[9.14228140091932e+2050488462815550  1.47925220414035e+2050488462815551]
```

The identity $\exp(A+B) = \exp(A)\exp(B)$ does not hold for noncommuting matrices:

```
>>> A = hilbert(3)
>>> B = A + eye(3)
>>> chop(mnorm(A*B - B*A))
0.0
>>> chop(mnorm(expm(A+B) - expm(A)*expm(B)))
0.0
>>> B = A + ones(3)
>>> mnorm(A*B - B*A)
1.8
>>> mnorm(expm(A+B) - expm(A)*expm(B))
42.0927851137247
```

mpmath.**cosm**(*ctx*, *A*)

Gives the cosine of a square matrix A, defined in analogy with the matrix exponential.

Examples:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> cosm(X)
[0.54030230586814               0.0               0.0]
[             0.0  0.54030230586814               0.0]
[             0.0               0.0  0.54030230586814]
>>> X = hilbert(3)
>>> cosm(X)
[ 0.424403834569555  -0.316643413047167  -0.221474945949293]
[-0.316643413047167   0.820646708837824  -0.127183694770039]
[-0.221474945949293  -0.127183694770039   0.909236687217541]
>>> X = matrix([[1+j,-2],[0,-j]])
>>> cosm(X)
[(0.833730025131149 - 0.988897705762865j)  (1.07485840848393 - 0.17192140544213j)]
[                                     0.0               (1.54308063481524 + 0.0j)]
```

mpmath.**sinm**(*ctx*, *A*)

Gives the sine of a square matrix A, defined in analogy with the matrix exponential.

Examples:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
```

```
>>> sinm(X)
[0.841470984807897                0.0                0.0]
[              0.0  0.841470984807897                0.0]
[              0.0                0.0  0.841470984807897]
>>> X = hilbert(3)
>>> sinm(X)
[0.711608512150994  0.339783913247439  0.220742837314741]
[0.339783913247439  0.244113865695532  0.187231271174372]
[0.220742837314741  0.187231271174372  0.155816730769635]
>>> X = matrix([[1+j,-2],[0,-j]])
>>> sinm(X)
[(1.29845758141598 + 0.634963914784736j)  (-1.96751511930922 + 0.314700021761367j)]
[                                     0.0                   (0.0 - 1.1752011936438j)]
```

mpmath.**sqrtm**(*ctx, A, _may_rotate=2*)

Computes a square root of the square matrix A, i.e. returns a matrix $B = A^{1/2}$ such that $B^2 = A$. The square root of a matrix, if it exists, is not unique.

Examples

Square roots of some simple matrices:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> sqrtm([[1,0], [0,1]])
[1.0  0.0]
[0.0  1.0]
>>> sqrtm([[0,0], [0,0]])
[0.0  0.0]
[0.0  0.0]
>>> sqrtm([[2,0],[0,1]])
[1.4142135623731  0.0]
[            0.0  1.0]
>>> sqrtm([[1,1],[1,0]])
[ (0.920442065259926 - 0.21728689675164j)  (0.568864481005783 + 0.351577584254143j)]
[(0.568864481005783 + 0.351577584254143j)  (0.351577584254143 - 0.568864481005783j)]
>>> sqrtm([[1,0],[0,1]])
[1.0  0.0]
[0.0  1.0]
>>> sqrtm([[-1,0],[0,1]])
[(0.0 - 1.0j)           0.0]
[         0.0  (1.0 + 0.0j)]
>>> sqrtm([[j,0],[0,j]])
[(0.707106781186547 + 0.707106781186547j)                                       0.0]
[                                      0.0  (0.707106781186547 + 0.707106781186547j)]
```

A square root of a rotation matrix, giving the corresponding half-angle rotation matrix:

```
>>> t1 = 0.75
>>> t2 = t1 * 0.5
>>> A1 = matrix([[cos(t1), -sin(t1)], [sin(t1), cos(t1)]])
>>> A2 = matrix([[cos(t2), -sin(t2)], [sin(t2), cos(t2)]])
>>> sqrtm(A1)
[0.930507621912314  -0.366272529086048]
[0.366272529086048   0.930507621912314]
>>> A2
[0.930507621912314  -0.366272529086048]
[0.366272529086048   0.930507621912314]
```

The identity $(A^2)^{1/2} = A$ does not necessarily hold:

```
>>> A = matrix([[4,1,4],[7,8,9],[10,2,11]])
>>> sqrtm(A**2)
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> sqrtm(A)**2
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> A = matrix([[-4,1,4],[7,-8,9],[10,2,11]])
>>> sqrtm(A**2)
[  7.43715112194995  -0.324127569985474   1.8481718827526]
[-0.251549715716942    9.32699765900402  2.48221180985147]
[  4.11609388833616   0.775751877098258   13.017955697342]
>>> chop(sqrtm(A)**2)
[-4.0   1.0   4.0]
[ 7.0  -8.0   9.0]
[10.0   2.0  11.0]
```

For some matrices, a square root does not exist:

```
>>> sqrtm([[0,1], [0,0]])
Traceback (most recent call last):
  ...
ZeroDivisionError: matrix is numerically singular
```

Two examples from the documentation for Matlab's `sqrtm`:

```
>>> mp.dps = 15; mp.pretty = True
>>> sqrtm([[7,10],[15,22]])
[1.56669890360128  1.74077655955698]
[2.61116483933547  4.17786374293675]
>>>
>>> X = matrix(\
...   [[5,-4,1,0,0],
...   [-4,6,-4,1,0],
...   [1,-4,6,-4,1],
...   [0,1,-4,6,-4],
...   [0,0,1,-4,5]])
>>> Y = matrix(\
...   [[2,-1,-0,-0,-0],
...   [-1,2,-1,0,-0],
...   [0,-1,2,-1,0],
...   [-0,0,-1,2,-1],
...   [-0,-0,-0,-1,2]])
>>> mnorm(sqrtm(X) - Y)
4.53155328326114e-19
```

`mpmath.`**`logm`**(*ctx*, *A*)

Computes a logarithm of the square matrix A, i.e. returns a matrix $B = \log(A)$ such that $\exp(B) = A$. The logarithm of a matrix, if it exists, is not unique.

Examples

Logarithms of some simple matrices:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> logm(X)
```

```
[0.0  0.0  0.0]
[0.0  0.0  0.0]
[0.0  0.0  0.0]
>>> logm(2*X)
[0.693147180559945                0.0                0.0]
[              0.0  0.693147180559945                0.0]
[              0.0                0.0  0.693147180559945]
>>> logm(expm(X))
[1.0  0.0  0.0]
[0.0  1.0  0.0]
[0.0  0.0  1.0]
```

A logarithm of a complex matrix:

```
>>> X = matrix([[2+j, 1, 3], [1-j, 1-2*j, 1], [-4, -5, j]])
>>> B = logm(X)
>>> nprint(B)
[ (0.808757 + 0.107759j)    (2.20752 + 0.202762j)   (1.07376 - 0.773874j)]
[ (0.905709 - 0.107795j)  (0.0287395 - 0.824993j)  (0.111619 + 0.514272j)]
[(-0.930151 + 0.399512j)   (-2.06266 - 0.674397j)  (0.791552 + 0.519839j)]
>>> chop(expm(B))
[(2.0 + 1.0j)           1.0           3.0]
[(1.0 - 1.0j)  (1.0 - 2.0j)           1.0]
[        -4.0          -5.0  (0.0 + 1.0j)]
```

A matrix X close to the identity matrix, for which $\log(\exp(X)) = \exp(\log(X)) = X$ holds:

```
>>> X = eye(3) + hilbert(3)/4
>>> X
[              1.25             0.125  0.0833333333333333]
[             0.125  1.08333333333333              0.0625]
[0.0833333333333333            0.0625                1.05]
>>> logm(expm(X))
[              1.25             0.125  0.0833333333333333]
[             0.125  1.08333333333333              0.0625]
[0.0833333333333333            0.0625                1.05]
>>> expm(logm(X))
[              1.25             0.125  0.0833333333333333]
[             0.125  1.08333333333333              0.0625]
[0.0833333333333333            0.0625                1.05]
```

A logarithm of a rotation matrix, giving back the angle of the rotation:

```
>>> t = 3.7
>>> A = matrix([[cos(t),sin(t)],[-sin(t),cos(t)]])
>>> chop(logm(A))
[             0.0  -2.58318530717959]
[2.58318530717959                0.0]
>>> (2*pi-t)
2.58318530717959
```

For some matrices, a logarithm does not exist:

```
>>> logm([[1,0], [0,0]])
Traceback (most recent call last):
  ...
ZeroDivisionError: matrix is numerically singular
```

Logarithm of a matrix with large entries:

```
>>> logm(hilbert(3) * 10**20).apply(re)
[ 45.5597513593433  1.27721006042799  0.317662687717978]
[ 1.27721006042799  42.5222778973542   2.24003708791604]
[0.317662687717978  2.24003708791604    42.395212822267]
```

mpmath.**powm**(*ctx, A, r*)

Computes $A^r = \exp(A \log r)$ for a matrix A and complex number r.

Examples

Powers and inverse powers of a matrix:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> A = matrix([[4,1,4],[7,8,9],[10,2,11]])
>>> powm(A, 2)
[ 63.0  20.0   69.0]
[174.0  89.0  199.0]
[164.0  48.0  179.0]
>>> chop(powm(powm(A, 4), 1/4.))
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> powm(extraprec(20)(powm)(A, -4), -1/4.)
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> chop(powm(powm(A, 1+0.5j), 1/(1+0.5j)))
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
>>> powm(extraprec(5)(powm)(A, -1.5), -1/(1.5))
[ 4.0  1.0   4.0]
[ 7.0  8.0   9.0]
[10.0  2.0  11.0]
```

A Fibonacci-generating matrix:

```
>>> powm([[1,1],[1,0]], 10)
[89.0  55.0]
[55.0  34.0]
>>> fib(10)
55.0
>>> powm([[1,1],[1,0]], 6.5)
[(16.5166626964253 - 0.0121089837381789j)  (10.2078589271083 + 0.0195927472575932j)]
[(10.2078589271083 + 0.0195927472575932j)  (6.30880376931698 - 0.0317017309957721j)]
>>> (phi**6.5 - (1-phi)**6.5)/sqrt(5)
(10.2078589271083 - 0.0195927472575932j)
>>> powm([[1,1],[1,0]], 6.2)
[ (14.3076953002666 - 0.008222855781077j)  (8.81733464837593 + 0.0133048601383712j)]
[(8.81733464837593 + 0.0133048601383712j)  (5.49036065189071 - 0.0215277159194482j)]
>>> (phi**6.2 - (1-phi)**6.2)/sqrt(5)
(8.81733464837593 - 0.0133048601383712j)
```

Number identification

Most function in mpmath are concerned with producing approximations from exact mathematical formulas. It is also useful to consider the inverse problem: given only a decimal

approximation for a number, such as 0.73205080756887729352744 63, is it possible to find an exact formula?

Subject to certain restrictions, such "reverse engineering" is indeed possible thanks to the existence of *integer relation algorithms*. Mpmath implements the PSLQ algorithm (developed by H. Ferguson), which is one such algorithm.

Automated number recognition based on PSLQ is not a silver bullet. Any occurring transcendental constants (π, e, etc) must be guessed by the user, and the relation between those constants in the formula must be linear (such as $x = 3\pi + 4e$). More complex formulas can be found by combining PSLQ with functional transformations; however, this is only feasible to a limited extent since the computation time grows exponentially with the number of operations that need to be combined.

The number identification facilities in mpmath are inspired by the Inverse Symbolic Calculator (ISC). The ISC is more powerful than mpmath, as it uses a lookup table of millions of precomputed constants (thereby mitigating the problem with exponential complexity).

Constant recognition

identify()

mpmath.**identify**(*ctx*, *x*, *constants*=[], *tol=None*, *maxcoeff=1000*, *full=False*, *verbose=False*)

Given a real number x, `identify(x)` attempts to find an exact formula for x. This formula is returned as a string. If no match is found, `None` is returned. With `full=True`, a list of matching formulas is returned.

As a simple example, `identify()` (page 928) will find an algebraic formula for the golden ratio:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> identify(phi)
'((1+sqrt(5))/2)'
```

`identify()` (page 928) can identify simple algebraic numbers and simple combinations of given base constants, as well as certain basic transformations thereof. More specifically, `identify()` (page 928) looks for the following:

1. Fractions
2. Quadratic algebraic numbers
3. Rational linear combinations of the base constants
4. Any of the above after first transforming x into $f(x)$ where $f(x)$ is $1/x$, $\sqrt{x}$, x^2, $\log x$ or $\exp x$, either directly or with x or $f(x)$ multiplied or divided by one of the base constants
5. Products of fractional powers of the base constants and small integers

Base constants can be given as a list of strings representing mpmath expressions (`identify()` (page 928) will `eval` the strings to numerical values and use the original strings for the output), or as a dict of formula:value pairs.

In order not to produce spurious results, `identify()` (page 928) should be used with high precision; preferably 50 digits or more.

Examples

Simple identifications can be performed safely at standard precision. Here the default recognition of rational, algebraic, and exp/log of algebraic numbers is demonstrated:

```
>>> mp.dps = 15
>>> identify(0.22222222222222222)
'(2/9)'
>>> identify(1.9662210973805663)
'sqrt(((24+sqrt(48))/8))'
>>> identify(4.1132503787829275)
'exp((sqrt(8)/2))'
>>> identify(0.881373587019543)
'log(((2+sqrt(8))/2))'
```

By default, `identify()` (page 928) does not recognize π. At standard precision it finds a not too useful approximation. At slightly increased precision, this approximation is no longer accurate enough and `identify()` (page 928) more correctly returns `None`:

```
>>> identify(pi)
'(2**(176/117)*3**(20/117)*5**(35/39))/(7**(92/117))'
>>> mp.dps = 30
>>> identify(pi)
>>>
```

Numbers such as π, and simple combinations of user-defined constants, can be identified if they are provided explicitly:

```
>>> identify(3*pi-2*e, ['pi', 'e'])
'(3*pi + (-2)*e)'
```

Here is an example using a dict of constants. Note that the constants need not be "atomic"; `identify()` (page 928) can just as well express the given number in terms of expressions given by formulas:

```
>>> identify(pi+e, {'a':pi+2, 'b':2*e})
'((-2) + 1*a + (1/2)*b)'
```

Next, we attempt some identifications with a set of base constants. It is necessary to increase the precision a bit.

```
>>> mp.dps = 50
>>> base = ['sqrt(2)','pi','log(2)']
>>> identify(0.25, base)
'(1/4)'
>>> identify(3*pi + 2*sqrt(2) + 5*log(2)/7, base)
'(2*sqrt(2) + 3*pi + (5/7)*log(2))'
>>> identify(exp(pi+2), base)
'exp((2 + 1*pi))'
>>> identify(1/(3+sqrt(2)), base)
'((3/7) + (-1/7)*sqrt(2))'
>>> identify(sqrt(2)/(3*pi+4), base)
'sqrt(2)/(4 + 3*pi)'
>>> identify(5**(mpf(1)/3)*pi*log(2)**2, base)
'5**(1/3)*pi*log(2)**2'
```

An example of an erroneous solution being found when too low precision is used:

```
>>> mp.dps = 15
>>> identify(1/(3*pi-4*e+sqrt(8)), ['pi', 'e', 'sqrt(2)'])
'((11/25) + (-158/75)*pi + (76/75)*e + (44/15)*sqrt(2))'
>>> mp.dps = 50
```

```
>>> identify(1/(3*pi-4*e+sqrt(8)), ['pi', 'e', 'sqrt(2)'])
'1/(3*pi + (-4)*e + 2*sqrt(2))'
```

Finding approximate solutions

The tolerance `tol` defaults to 3/4 of the working precision. Lowering the tolerance is useful for finding approximate matches. We can for example try to generate approximations for pi:

```
>>> mp.dps = 15
>>> identify(pi, tol=1e-2)
'(22/7)'
>>> identify(pi, tol=1e-3)
'(355/113)'
>>> identify(pi, tol=1e-10)
'(5**(339/269))/(2**(64/269)*3**(13/269)*7**(92/269))'
```

With `full=True`, and by supplying a few base constants, `identify` can generate almost endless lists of approximations for any number (the output below has been truncated to show only the first few):

```
>>> for p in identify(pi, ['e', 'catalan'], tol=1e-5, full=True):
...     print(p)
...
e/log((6 + (-4/3)*e))
(3**3*5*e*catalan**2)/(2*7**2)
sqrt(((-13) + 1*e + 22*catalan))
log(((-6) + 24*e + 4*catalan)/e)
exp(catalan*((-1/5) + (8/15)*e))
catalan*(6 + (-6)*e + 15*catalan)
sqrt((5 + 26*e + (-3)*catalan))/e
e*sqrt(((-27) + 2*e + 25*catalan))
log(((-1) + (-11)*e + 59*catalan))
((3/20) + (21/20)*e + (3/20)*catalan)
...
```

The numerical values are roughly as close to π as permitted by the specified tolerance:

```
>>> e/log(6-4*e/3)
3.14157719846001
>>> 135*e*catalan**2/98
3.14166950419369
>>> sqrt(e-13+22*catalan)
3.14158000062992
>>> log(24*e-6+4*catalan)-1
3.14158791577159
```

Symbolic processing

The output formula can be evaluated as a Python expression. Note however that if fractions (like '2/3') are present in the formula, Python's `eval()` may erroneously perform integer division. Note also that the output is not necessarily in the algebraically simplest form:

```
>>> identify(sqrt(2))
'(sqrt(8)/2)'
```

As a solution to both problems, consider using SymPy's `sympify()` to convert the formula into a symbolic expression. SymPy can be used to pretty-print or further simplify the formula symbolically:

```
>>> from sympy import sympify
>>> sympify(identify(sqrt(2)))
sqrt(2)
```

Sometimes identify() (page 928) can simplify an expression further than a symbolic algorithm:

```
>>> from sympy import simplify
>>> x = sympify('-1/(-3/2+(1/2)*sqrt(5))*sqrt(3/2-1/2*sqrt(5))')
>>> x
1/sqrt(3/2 - sqrt(5)/2)
>>> x = simplify(x)
>>> x
2/sqrt(6 - 2*sqrt(5))
>>> mp.dps = 30
>>> x = sympify(identify(x.evalf(30)))
>>> x
1/2 + sqrt(5)/2
```

(In fact, this functionality is available directly in SymPy as the function nsimplify(), which is essentially a wrapper for identify() (page 928).)

Miscellaneous issues and limitations

The input x must be a real number. All base constants must be positive real numbers and must not be rationals or rational linear combinations of each other.

The worst-case computation time grows quickly with the number of base constants. Already with 3 or 4 base constants, identify() (page 928) may require several seconds to finish. To search for relations among a large number of constants, you should consider using pslq() (page 933) directly.

The extended transformations are applied to x, not the constants separately. As a result, identify will for example be able to recognize exp(2*pi+3) with pi given as a base constant, but not 2*exp(pi)+3. It will be able to recognize the latter if exp(pi) is given explicitly as a base constant.

Algebraic identification

findpoly()

mpmath.**findpoly**(*ctx*, *x*, *n=1*, ***kwargs*)

findpoly(x, n) returns the coefficients of an integer polynomial P of degree at most n such that $P(x) \approx 0$. If no polynomial having x as a root can be found, findpoly() (page 931) returns None.

findpoly() (page 931) works by successively calling pslq() (page 933) with the vectors $[1, x]$, $[1, x, x^2]$, $[1, x, x^2, x^3]$, ..., $[1, x, x^2, .., x^n]$ as input. Keyword arguments given to findpoly() (page 931) are forwarded verbatim to pslq() (page 933). In particular, you can specify a tolerance for $P(x)$ with tol and a maximum permitted coefficient size with maxcoeff.

For large values of n, it is recommended to run findpoly() (page 931) at high precision; preferably 50 digits or more.

Examples

By default (degree $n = 1$), findpoly() (page 931) simply finds a linear polynomial with a rational root:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> findpoly(0.7)
[-10, 7]
```

The generated coefficient list is valid input to `polyval` and `polyroots`:

```
>>> nprint(polyval(findpoly(phi, 2), phi), 1)
-2.0e-16
>>> for r in polyroots(findpoly(phi, 2)):
...     print(r)
...
-0.618033988749895
1.61803398874989
```

Numbers of the form $m + n\sqrt{p}$ for integers (m, n, p) are solutions to quadratic equations. As we find here, $1 + \sqrt{2}$ is a root of the polynomial $x^2 - 2x - 1$:

```
>>> findpoly(1+sqrt(2), 2)
[1, -2, -1]
>>> findroot(lambda x: x**2 - 2*x - 1, 1)
2.4142135623731
```

Despite only containing square roots, the following number results in a polynomial of degree 4:

```
>>> findpoly(sqrt(2)+sqrt(3), 4)
[1, 0, -10, 0, 1]
```

In fact, $x^4 - 10x^2 + 1$ is the *minimal polynomial* of $r = \sqrt{2} + \sqrt{3}$, meaning that a rational polynomial of lower degree having r as a root does not exist. Given sufficient precision, `findpoly()` (page 931) will usually find the correct minimal polynomial of a given algebraic number.

Non-algebraic numbers

If `findpoly()` (page 931) fails to find a polynomial with given coefficient size and tolerance constraints, that means no such polynomial exists.

We can verify that π is not an algebraic number of degree 3 with coefficients less than 1000:

```
>>> mp.dps = 15
>>> findpoly(pi, 3)
>>>
```

It is always possible to find an algebraic approximation of a number using one (or several) of the following methods:

1. Increasing the permitted degree
2. Allowing larger coefficients
3. Reducing the tolerance

One example of each method is shown below:

```
>>> mp.dps = 15
>>> findpoly(pi, 4)
[95, -545, 863, -183, -298]
>>> findpoly(pi, 3, maxcoeff=10000)
[836, -1734, -2658, -457]
```

```
>>> findpoly(pi, 3, tol=1e-7)
[-4, 22, -29, -2]
```

It is unknown whether Euler's constant is transcendental (or even irrational). We can use `findpoly()` (page 931) to check that if is an algebraic number, its minimal polynomial must have degree at least 7 and a coefficient of magnitude at least 1000000:

```
>>> mp.dps = 200
>>> findpoly(euler, 6, maxcoeff=10**6, tol=1e-100, maxsteps=1000)
>>>
```

Note that the high precision and strict tolerance is necessary for such high-degree runs, since otherwise unwanted low-accuracy approximations will be detected. It may also be necessary to set maxsteps high to prevent a premature exit (before the coefficient bound has been reached). Running with `verbose=True` to get an idea what is happening can be useful.

Integer relations (PSLQ)

pslq()

mpmath.**pslq**(*ctx, x, tol=None, maxcoeff=1000, maxsteps=100, verbose=False*)

Given a vector of real numbers $x = [x_0, x_1, ..., x_n]$, `pslq(x)` uses the PSLQ algorithm to find a list of integers $[c_0, c_1, ..., c_n]$ such that

$$|c_1 x_1 + c_2 x_2 + ... + c_n x_n| < \text{tol}$$

and such that $\max |c_k| <$ maxcoeff. If no such vector exists, `pslq()` (page 933) returns `None`. The tolerance defaults to 3/4 of the working precision.

Examples

Find rational approximations for π:

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> pslq([-1, pi], tol=0.01)
[22, 7]
>>> pslq([-1, pi], tol=0.001)
[355, 113]
>>> mpf(22)/7; mpf(355)/113; +pi
3.14285714285714
3.14159292035398
3.14159265358979
```

Pi is not a rational number with denominator less than 1000:

```
>>> pslq([-1, pi])
>>>
```

To within the standard precision, it can however be approximated by at least one rational number with denominator less than 10^{12}:

```
>>> p, q = pslq([-1, pi], maxcoeff=10**12)
>>> print(p); print(q)
238410049439
75888275702
>>> mpf(p)/q
3.14159265358979
```

The PSLQ algorithm can be applied to long vectors. For example, we can investigate the rational (in)dependence of integer square roots:

```
>>> mp.dps = 30
>>> pslq([sqrt(n) for n in range(2, 5+1)])
>>>
>>> pslq([sqrt(n) for n in range(2, 6+1)])
>>>
>>> pslq([sqrt(n) for n in range(2, 8+1)])
[2, 0, 0, 0, 0, 0, -1]
```

Machin formulas

A famous formula for π is Machin's,

$$\frac{\pi}{4} = 4\,\mathrm{acot}\,5 - \mathrm{acot}\,239$$

There are actually infinitely many formulas of this type. Two others are

$$\frac{\pi}{4} = \mathrm{acot}\,1$$

$$\frac{\pi}{4} = 12\,\mathrm{acot}\,49 + 32\,\mathrm{acot}\,57 + 5\,\mathrm{acot}\,239 + 12\,\mathrm{acot}\,110443$$

We can easily verify the formulas using the PSLQ algorithm:

```
>>> mp.dps = 30
>>> pslq([pi/4, acot(1)])
[1, -1]
>>> pslq([pi/4, acot(5), acot(239)])
[1, -4, 1]
>>> pslq([pi/4, acot(49), acot(57), acot(239), acot(110443)])
[1, -12, -32, 5, -12]
```

We could try to generate a custom Machin-like formula by running the PSLQ algorithm with a few inverse cotangent values, for example acot(2), acot(3) ... acot(10). Unfortunately, there is a linear dependence among these values, resulting in only that dependence being detected, with a zero coefficient for π:

```
>>> pslq([pi] + [acot(n) for n in range(2,11)])
[0, 1, -1, 0, 0, 0, -1, 0, 0, 0]
```

We get better luck by removing linearly dependent terms:

```
>>> pslq([pi] + [acot(n) for n in range(2,11) if n not in (3, 5)])
[1, -8, 0, 0, 4, 0, 0, 0]
```

In other words, we found the following formula:

```
>>> 8*acot(2) - 4*acot(7)
3.141592653589793238462643383 28
>>> +pi
3.141592653589793238462643383 28
```

Algorithm

This is a fairly direct translation to Python of the pseudocode given by David Bailey, "The PSLQ Integer Relation Algorithm": http://www.cecm.sfu.ca/organics/papers/bailey/paper/html/node3.html

The present implementation uses fixed-point instead of floating-point arithmetic, since this is significantly (about 7x) faster.

5.15.4 End matter

Precision and representation issues

Most of the time, using mpmath is simply a matter of setting the desired precision and entering a formula. For verification purposes, a quite (but not always!) reliable technique is to calculate the same thing a second time at a higher precision and verifying that the results agree.

To perform more advanced calculations, it is important to have some understanding of how mpmath works internally and what the possible sources of error are. This section gives an overview of arbitrary-precision binary floating-point arithmetic and some concepts from numerical analysis.

The following concepts are important to understand:

- The main sources of numerical errors are rounding and cancellation, which are due to the use of finite-precision arithmetic, and truncation or approximation errors, which are due to approximating infinite sequences or continuous functions by a finite number of samples.
- Errors propagate between calculations. A small error in the input may result in a large error in the output.
- Most numerical algorithms for complex problems (e.g. integrals, derivatives) give wrong answers for sufficiently ill-behaved input. Sometimes virtually the only way to get a wrong answer is to design some very contrived input, but at other times the chance of accidentally obtaining a wrong result even for reasonable-looking input is quite high.
- Like any complex numerical software, mpmath has implementation bugs. You should be reasonably suspicious about any results computed by mpmath, even those it claims to be able to compute correctly! If possible, verify results analytically, try different algorithms, and cross-compare with other software.

Precision, error and tolerance

The following terms are common in this documentation:

- *Precision* (or *working precision*) is the precision at which floating-point arithmetic operations are performed.
- *Error* is the difference between a computed approximation and the exact result.
- *Accuracy* is the inverse of error.
- *Tolerance* is the maximum error (or minimum accuracy) desired in a result.

Error and accuracy can be measured either directly, or logarithmically in bits or digits. Specifically, if a $\hat{y}$ is an approximation for y, then

- (Direct) absolute error = $|\hat{y} - y|$
- (Direct) relative error = $|\hat{y} - y||y|^{-1}$
- (Direct) absolute accuracy = $|\hat{y} - y|^{-1}$
- (Direct) relative accuracy = $|\hat{y} - y|^{-1}|y|$
- (Logarithmic) absolute error = $\log_b |\hat{y} - y|$
- (Logarithmic) relative error = $\log_b |\hat{y} - y| - \log_b |y|$
- (Logarithmic) absolute accuracy = $-\log_b |\hat{y} - y|$

- (Logarithmic) relative accuracy = $-\log_b |\hat{y} - y| + \log_b |y|$

where $b = 2$ and $b = 10$ for bits and digits respectively. Note that:

- The logarithmic error roughly equals the position of the first incorrect bit or digit
- The logarithmic accuracy roughly equals the number of correct bits or digits in the result

These definitions also hold for complex numbers, using $|a + bi| = \sqrt{a^2 + b^2}$.

Full accuracy means that the accuracy of a result at least equals *prec*-1, i.e. it is correct except possibly for the last bit.

Representation of numbers

Mpmath uses binary arithmetic. A binary floating-point number is a number of the form $man \times 2^{exp}$ where both *man* (the *mantissa*) and *exp* (the *exponent*) are integers. Some examples of floating-point numbers are given in the following table.

Number	Mantissa	Exponent
3	3	0
10	5	1
-16	-1	4
1.25	5	-2

The representation as defined so far is not unique; one can always multiply the mantissa by 2 and subtract 1 from the exponent with no change in the numerical value. In mpmath, numbers are always normalized so that *man* is an odd number, with the exception of zero which is always taken to have *man = exp = 0*. With these conventions, every representable number has a unique representation. (Mpmath does not currently distinguish between positive and negative zero.)

Simple mathematical operations are now easy to define. Due to uniqueness, equality testing of two numbers simply amounts to separately checking equality of the mantissas and the exponents. Multiplication of nonzero numbers is straightforward: $(m2^e) \times (n2^f) = (mn) \times 2^{e+f}$. Addition is a bit more involved: we first need to multiply the mantissa of one of the operands by a suitable power of 2 to obtain equal exponents.

More technically, mpmath represents a floating-point number as a 4-tuple *(sign, man, exp, bc)* where *sign* is 0 or 1 (indicating positive vs negative) and the mantissa is nonnegative; *bc* (*bitcount*) is the size of the absolute value of the mantissa as measured in bits. Though redundant, keeping a separate sign field and explicitly keeping track of the bitcount significantly speeds up arithmetic (the bitcount, especially, is frequently needed but slow to compute from scratch due to the lack of a Python built-in function for the purpose).

Contrary to popular belief, floating-point *numbers* do not come with an inherent "small uncertainty", although floating-point *arithmetic* generally is inexact. Every binary floating-point number is an exact rational number. With arbitrary-precision integers used for the mantissa and exponent, floating-point numbers can be added, subtracted and multiplied *exactly*. In particular, integers and integer multiples of 1/2, 1/4, 1/8, 1/16, etc. can be represented, added and multiplied exactly in binary floating-point arithmetic.

Floating-point arithmetic is generally approximate because the size of the mantissa must be limited for efficiency reasons. The maximum allowed width (bitcount) of the mantissa is called the precision or *prec* for short. Sums and products of floating-point numbers are exact as long as the absolute value of the mantissa is smaller than 2^{prec}. As soon as the mantissa becomes larger than this, it is truncated to contain at most *prec* bits (the exponent is incremented accordingly to preserve the magnitude of the number), and this operation introduces a rounding error. Division is also generally inexact; although we can add and

multiply exactly by setting the precision high enough, no precision is high enough to represent for example 1/3 exactly (the same obviously applies for roots, trigonometric functions, etc).

The special numbers `+inf`, `-inf` and `nan` are represented internally by a zero mantissa and a nonzero exponent.

Mpmath uses arbitrary precision integers for both the mantissa and the exponent, so numbers can be as large in magnitude as permitted by the computer's memory. Some care may be necessary when working with extremely large numbers. Although standard arithmetic operators are safe, it is for example futile to attempt to compute the exponential function of of 10^{100000}. Mpmath does not complain when asked to perform such a calculation, but instead chugs away on the problem to the best of its ability, assuming that computer resources are infinite. In the worst case, this will be slow and allocate a huge amount of memory; if entirely impossible Python will at some point raise `OverflowError: long int too large to convert to int`.

For further details on how the arithmetic is implemented, refer to the mpmath source code. The basic arithmetic operations are found in the `libmp` directory; many functions there are commented extensively.

Decimal issues

Mpmath uses binary arithmetic internally, while most interaction with the user is done via the decimal number system. Translating between binary and decimal numbers is a somewhat subtle matter; many Python novices run into the following "bug" (addressed in the General Python FAQ):

```
>>> 0.1
0.10000000000000001
```

Decimal fractions fall into the category of numbers that generally cannot be represented exactly in binary floating-point form. For example, none of the numbers 0.1, 0.01, 0.001 has an exact representation as a binary floating-point number. Although mpmath can approximate decimal fractions with any accuracy, it does not solve this problem for all uses; users who need *exact* decimal fractions should look at the `decimal` module in Python's standard library (or perhaps use fractions, which are much faster).

With *prec* bits of precision, an arbitrary number can be approximated relatively to within 2^{-prec}, or within 10^{-dps} for *dps* decimal digits. The equivalent values for *prec* and *dps* are therefore related proportionally via the factor $C = \log(10)/\log(2)$, or roughly 3.32. For example, the standard (binary) precision in mpmath is 53 bits, which corresponds to a decimal precision of 15.95 digits.

More precisely, mpmath uses the following formulas to translate between *prec* and *dps*:

```
dps(prec) = max(1, int(round(int(prec) / C - 1)))

prec(dps) = max(1, int(round((int(dps) + 1) * C)))
```

Note that the dps is set 1 decimal digit lower than the corresponding binary precision. This is done to hide minor rounding errors and artifacts resulting from binary-decimal conversion. As a result, mpmath interprets 53 bits as giving 15 digits of decimal precision, not 16.

The *dps* value controls the number of digits to display when printing numbers with `str()`, while the decimal precision used by `repr()` is set two or three digits higher. For example, with 15 dps we have:

```
>>> from mpmath import *
>>> mp.dps = 15
>>> str(pi)
'3.14159265358979'
>>> repr(+pi)
"mpf('3.1415926535897931')"
```

The extra digits in the output from `repr` ensure that `x == eval(repr(x))` holds, i.e. that numbers can be converted to strings and back losslessly.

It should be noted that precision and accuracy do not always correlate when translating between binary and decimal. As a simple example, the number 0.1 has a decimal precision of 1 digit but is an infinitely accurate representation of 1/10. Conversely, the number 2^{-50} has a binary representation with 1 bit of precision that is infinitely accurate; the same number can actually be represented exactly as a decimal, but doing so requires 35 significant digits:

```
0.00000000000000088817841970012523233890533447265625
```

All binary floating-point numbers can be represented exactly as decimals (possibly requiring many digits), but the converse is false.

Correctness guarantees

Basic arithmetic operations (with the `mp` context) are always performed with correct rounding. Results that can be represented exactly are guranteed to be exact, and results from single inexact operations are guaranteed to be the best possible rounded values. For higher-level operations, mpmath does not generally guarantee correct rounding. In general, mpmath only guarantees that it will use at least the user-set precision to perform a given calculation. *The user may have to manually set the working precision higher than the desired accuracy for the result, possibly much higher.*

Functions for evaluation of transcendental functions, linear algebra operations, numerical integration, etc., usually automatically increase the working precision and use a stricter tolerance to give a correctly rounded result with high probability: for example, at 50 bits the temporary precision might be set to 70 bits and the tolerance might be set to 60 bits. It can often be assumed that such functions return values that have full accuracy, given inputs that are exact (or sufficiently precise approximations of exact values), but the user must exercise judgement about whether to trust mpmath.

The level of rigor in mpmath covers the entire spectrum from "always correct by design" through "nearly always correct" and "handling the most common errors" to "just computing blindly and hoping for the best". Of course, a long-term development goal is to successively increase the rigor where possible. The following list might give an idea of the current state.

Operations that are correctly rounded:

- Addition, subtraction and multiplication of real and complex numbers.
- Division and square roots of real numbers.
- Powers of real numbers, assuming sufficiently small integer exponents (huge powers are rounded in the right direction, but possibly farther than necessary).
- Conversion from decimal to binary, for reasonably sized numbers (roughly between 10^{-100} and 10^{100}).
- Typically, transcendental functions for exact input-output pairs.

Operations that should be fully accurate (however, the current implementation may be based on a heuristic error analysis):

- Radix conversion (large or small numbers).
- Mathematical constants like π.
- Both real and imaginary parts of exp, cos, sin, cosh, sinh, log.
- Other elementary functions (the largest of the real and imaginary part).
- The gamma and log-gamma functions (the largest of the real and the imaginary part; both, when close to real axis).
- Some functions based on hypergeometric series (the largest of the real and imaginary part).

Correctness of root-finding, numerical integration, etc. largely depends on the well-behavedness of the input functions. Specific limitations are sometimes noted in the respective sections of the documentation.

Double precision emulation

On most systems, Python's `float` type represents an IEEE 754 *double precision* number, with a precision of 53 bits and rounding-to-nearest. With default precision (`mp.prec = 53`), the mpmath `mpf` type roughly emulates the behavior of the `float` type. Sources of incompatibility include the following:

- In hardware floating-point arithmetic, the size of the exponent is restricted to a fixed range: regular Python floats have a range between roughly 10^{-300} and 10^{300}. Mpmath does not emulate overflow or underflow when exponents fall outside this range.
- On some systems, Python uses 80-bit (extended double) registers for floating-point operations. Due to double rounding, this makes the `float` type less accurate. This problem is only known to occur with Python versions compiled with GCC on 32-bit systems.
- Machine floats very close to the exponent limit round subnormally, meaning that they lose accuracy (Python may raise an exception instead of rounding a `float` subnormally).
- Mpmath is able to produce more accurate results for transcendental functions.

Further reading

There are many excellent textbooks on numerical analysis and floating-point arithmetic. Some good web resources are:

- David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic
- The Wikipedia article about numerical analysis

References

The following is a non-comprehensive list of works used in the development of mpmath or cited for examples or mathematical definitions used in this documentation. References not listed here can be found in the source code.

5.16 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.

5.16.1 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 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 $\deg(r) < q$. For polynomials in one variables 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()
   2
5*x  + 10*x + 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
--- - --- + 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, be 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')
>>> q = Symbol('q')

>>> solve(x**2 + p*x + q, x)
         __________           __________
        /  2                 /  2
   p   \/  p  - 4*q     p   \/  p  - 4*q
[- - - -------------, - - + -------------]
   2         2          2         2

>>> solve_poly_system([y - x, x - 5], x, y)
[(5, 5)]

>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)

                              1   \/ 3 *I         1   \/ 3 *I
[(0, -I), (0, I), (1, 0), (- - - -------, 0), (- - + -------, 0)]
                              2      2            2      2
```

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 1626)). 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
```

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*h` and `g*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  9  2         19
-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  - 24⋅x⋅y
```

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 we 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)
        ⎛ 2         ⎞⋅⎛ 4    3    2         ⎞⋅⎛ 8    7    5    4    3         ⎞
(x - 1)⋅⎝x  + x + 1⎠ ⎝x  + x  + x  + x + 1⎠ ⎝x  - x  + x  - x  + x - 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)
  ⎛     3⋅i⎞ ⎛     3⋅i⎞
4⋅⎝x - ---⎠⋅⎝x + ---⎠⋅(x + 1 - 4⋅i)⋅(x + 1 + 4⋅i)
  ⎝      2⎠ ⎝      2⎠
```

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)
```

$$4\cdot\left(x - \frac{3\cdot i}{2}\right)\cdot\left(x + \frac{3\cdot i}{2}\right)\cdot(x + 1 - 4\cdot i)\cdot(x + 1 + 4\cdot i)$$

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`. Do do this we employ `cancel()` function:

```
>>> cancel(f/g)
```

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

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)
```

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

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))
```

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

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)
  ```

  $$\left(x^2 - x - 1\right)\cdot\left(x^2 + x - 1\right)$$

- 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 - 1 - α)·(x - α)·(x + α)·(x + 1 + α)
```

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))
        ⎛      ⎛          ___  ⎞⎞ ⎛      ⎛          ___  ⎞⎞
        ⎜      ⎜  1   \/ 3 ·i⎟⎟ ⎜      ⎜  1   \/ 3 ·i⎟⎟
(x + y)·⎜x + y·⎜- - - -------⎟⎟·⎜x + y·⎜- - + -------⎟⎟
        ⎝      ⎝  2      2   ⎠⎠ ⎝      ⎝  2      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          2
                (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*, **gens*, ***args*)
Return the degree of f in the given variable.

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
```

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 (mod 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.

Examples

```
>>> from sympy import invert
>>> 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
```

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.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 170), sympy.core.exprtools.factor_terms (page 171)

Examples

```
>>> from sympy import terms_gcd, cos, pi
>>> 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 `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, y

>>> 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`.

Examples

```
>>> from sympy import gff_list, ff
>>> from sympy.abc import x

>>> f = x**5 + 2*x**4 - x**3 - 2*x**2
```

```
>>> gff_list(f)
[(x, 1), (x + 2, 4)]

>>> (ff(x, 1)*ff(x + 2, 4)).expand() == 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) = 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 965) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, `factor()` (page 961) 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 961) 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 272)

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, error=False*)

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, ..., g_n)`, computes a set of quotients `q = (q_1, ..., q_n)` and the remainder `r` such that `f = q_1*f_1 + ... + q_n*f_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 $solve_{p}oly_{s}ystem()$.

References

1. [Buchberger01] (page 1627)
2. [Cox97] (page 1626)

[Buchberger01] (page 1627), [Cox97] (page 1626)

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 polynomial expressions.

`EC`(*f*, *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(*f, 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(*f, 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(*f, 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(*f, order=None*)

Returns the leading 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).LM()
x**2*y**0
```

LT(*f, order=None*)

Returns the leading 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).LT()
(x**2*y**0, 4)
```

TC(*f*)

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(*f*)

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(*f, 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(*f, 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(*f*)

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(*f*)

Returns all monomials from a univariate polynomial f.

See Also:

all_terms (page 968)

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(*f, 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()
[RootOf(x**3 + x + 1, 0),
 RootOf(x**3 + x + 1, 1),
 RootOf(x**3 + x + 1, 2)]
```

all_terms(*f*)

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(*f, 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(*f, *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(*f, native=False*)

Switch to a `list` representation.

cancel(*f, 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(*f, 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(*f, monom*)

Returns the coefficient of `monom` in `f` if there, else None.

See Also:

`nth` **(page 987)** 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(*f*, *order=None*)

Returns all non-zero coefficients from f in lex order.

See Also:

all_coeffs (page 968), coeff_monomial (page 970), nth (page 987)

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

cofactors(*f*, *g*)

Returns the GCD of f and g and their cofactors.

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 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(*f*, *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(*f*)

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(*f, 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(*f*)

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(*f*)

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(*f, gen=0*)

Returns degree of `f` in `x_j`.

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
```

degree_list(*f*)

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(*f, *specs*)

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(*f*)

Computes the discriminant of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

div(*f, 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(*f, *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(*f, 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(*f*)
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(*f, 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(*f, 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(*f*)

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(*f*)

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

>>> Poly(x**2 + 1).free_symbols
set([x])
>>> Poly(x**2 + y).free_symbols
set([x, y])
>>> Poly(x**2 + y, x).free_symbols
set([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
set([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(*f, 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(*f, 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(*f*)
Get the ground domain of `f`.

get_modulus(*f*)
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(*f*)

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(*f*)

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(*f, g, auto=True*)

Half extended Euclidean algorithm of `f` and `g`.

Returns `(s, h)` such that `h = gcd(f, g)` and `s*f = h (mod 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(*f, *gens*)

Return `True` if `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(*f*)

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 981).

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
```

inject(*f, 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(*f, *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(*f, all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)

Compute isolating intervals for roots of `f`.

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)]
```

invert(*f, 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 polnomial.

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 979).

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(*f*)

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(*f, 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(*f*)

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(*f*)

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(*f, gen*)

Remove dummy generators from the "left" 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')
```

max_norm(*f*)

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(*f, 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(*f, order=None*)

Returns all non-zero monomials from f in lex order.

See Also:

all_monoms (page 968)

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(*f, 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(*f, 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(*f*)

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 965) instance from raw representation.

nroots(*f, n=15, maxsteps=50, cleanup=True, error=False*)

Compute numerical approximations of roots of f.

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(*f, *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 970)

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(*f, 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(*f, 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(*f, 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(*f, 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(*f*, *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(*f*, *g*)

Polynomial pseudo-quotient of `f` by `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(*f*, *g*)

Polynomial pseudo-remainder of `f` by `g`.

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(*f*)

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(*f, 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(*f, 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(*f, 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(*f, 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()
[RootOf(x**3 + x + 1, 0)]
```

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 Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

rem(*f, g, auto=True*)

Computes the polynomial remainder of `f` by `g`.

Examples

```
>>> from sympy import Poly, ZZ, QQ
>>> 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(*f, *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(*f, 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(*f, 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 955) 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(*f, field=None*)

Recalculate the ground domain of a polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> 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(*f, n*)

Compute f**(-1) mod x**n.

root(*f, 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)
RootOf(x**3 - x**2 + 1, 0)
```

set_domain(*f, domain*)
Set the ground domain of `f`.

set_modulus(*f, 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(*f, 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(*f, x, m, n=None*)
Take a continuous subsequence of terms of `f`.

sqf_list(*f, 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(*f, 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(*f*)

Computes square-free norm of `f`.

Returns `s`, `f`, `r`, such that `g(x) = f(x-sa)` and `r(x) = 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(*f*)

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(*f*)

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(*f, 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(*f, 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(*f, 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(*f, 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(*f, order=None*)

Returns all non-zero terms from `f` in lex order.

See Also:

`all_terms` (page 968)

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), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

terms_gcd(*f*)

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(*f, 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):
...     return coeff//10**(2-k)

>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

to_exact(*f*)

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(*f*)

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(*f*)

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(*f*)

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
```

trunc(*f, 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(*f, g*)

Make f and g belong to the same domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> f, g = Poly(x/2 + 1), 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
set([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, ..., g_n)`, computes a set of quotients `q = (q_1, ..., q_n)` and the remainder `r` such that `f = q_1*f_1 + ... + 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, ..., x_n)`, then `f = f(x_{i_1}, x_{i_2}, ..., x_{i_n})`, where `(i_1, i_2, ..., i_n)` is a permutation of `(1, 2, ..., n)` (an element of the group `S_n`).

Returns a tuple of symmetric polynomials `(f1, f2, ..., fn)` such that `f = f1 + f2 + ... + fn`.

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*, ***args*)

Computes the minimal polynomial of an algebraic number.

Parameters **ex** : algebraic number expression

x : indipendent variable of the minimal polynomial

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.

Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve
>>> from sympy.abc import x

>>> minimal_polynomial(sqrt(2), x)
x**2 - 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
```

Options

compose : if `True` _minpoly1‘‘ is used, else the `groebner` algorithm

polys : if `True` returns a `Poly` object

sympy.polys.numberfields.**minpoly**(*ex*, *x=None*, ***args*)

Computes the minimal polynomial of an algebraic number.

Parameters **ex** : algebraic number expression

x : indipendent variable of the minimal polynomial

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.

Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve
>>> from sympy.abc import x
```

```
>>> minimal_polynomial(sqrt(2), x)
x**2 - 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
```

Options

compose : if `True` _minpoly1‘‘ is used, else the `groebner` algorithm

polys : if `True` returns a `Poly` object

`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

Orderings of monomials

Formal manipulation of roots of polynomials

class `sympy.polys.rootoftools.`**`RootOf`**
Represents k-th root of a univariate polynomial.

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.

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 tuple containing all those roots set the `multiple` flag to True.

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*, ***args*)

Generates n-th Swinnerton-Dyer polynomial in x.

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*, ***args*)

Generates cyclotomic polynomial of order n in x.

sympy.polys.specialpolys.**symmetric_poly**(*n*, **gens*, ***args*)

Generates symmetric polynomial of order n.

sympy.polys.specialpolys.**random_poly**(*x*, *n*, *inf*, *sup*, *domain=ZZ*, *polys=False*)
Return a polynomial of degree `n` with coefficients in `[inf, sup]`.

Orthogonal polynomials

sympy.polys.orthopolys.**chebyshevt_poly**(*n*, *x=None*, ***args*)
Generates Chebyshev polynomial of the first kind of degree n in x.

sympy.polys.orthopolys.**chebyshevu_poly**(*n*, *x=None*, ***args*)
Generates Chebyshev polynomial of the second kind of degree n in x.

sympy.polys.orthopolys.**gegenbauer_poly**(*n*, *a*, *x=None*, ***args*)
Generates Gegenbauer polynomial of degree n in x.

sympy.polys.orthopolys.**hermite_poly**(*n*, *x=None*, ***args*)
Generates Hermite polynomial of degree n in x.

sympy.polys.orthopolys.**jacobi_poly**(*n*, *a*, *b*, *x=None*, ***args*)
Generates Jacobi polynomial of degree n in x.

sympy.polys.orthopolys.**legendre_poly**(*n*, *x=None*, ***args*)
Generates Legendre polynomial of degree n in x.

sympy.polys.orthopolys.**laguerre_poly**(*n*, *x=None*, *alpha=None*, ***args*)
Generates Laguerre polynomial of degree n in x.

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 1006) 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 1006) is a complement to `apart()`, so `apart(together(expr))` should return expr unchanged. Note however, that `together()` (page 1006) uses only symbolic methods, so it might be necessary to use `cancel()` to perform algebraic simplification and minimise 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**(*expr, *args, **kwargs*)

Compute partial fraction decomposition of a rational function.

Given a rational function `f` compute partial fraction decomposition of `f`. Two algorithms are available: one is based on undetermined coefficients method and the other is Bronstein's full partial fraction decomposition algorithm.

See Also:

`apart_list` (page 1007), `assemble_partfrac_list` (page 1009)

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)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*_a*y/3 - y/3)/(-_a + x)))
```

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.

On can always rebuild a plain expression by using the function `assemble_partfrac_list`.

See Also:

`apart` (page 1007), `assemble_partfrac_list` (page 1009)

References

1. [Bronstein93] (page 1626)

[Bronstein93] (page 1626)

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 1007), `apart_list` (page 1007)

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),

>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

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 1627). An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008] (page 1627).

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
𝕆[x, y, order=ilex]
-------------------
    ⎛   3    2⎞
    ⎝- x  + y ⎠
```

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
    ⎛   3    2⎞
x + ⎝- x  + y ⎠

>>> 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	

`Ring.`**`free_module`**(*rank*)

Generate a free module of rank `rank` over self.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ[x].free_module(2)
QQ[x]**2
```

`Ring.`**`ideal`**(**gens*)

Generate an ideal of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ[x].ideal(x**2)
<x**2>
```

`Ring.`**`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[x].quotient_ring(QQ[x].ideal(x**2))
QQ[x]/<x**2>
>>> QQ[x].quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ[x]/[x**2]
QQ[x]/<x**2>
```

sympy.polys.domains.polynomialring.**PolynomialRing**(*ring*)

A class for representing multivariate polynomial rings.

Attributes

alias	
dtype	
one	
rep	
zero	

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[x].ideal(x**3 + 1)
>>> QQ[x].quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ[x]/I
QQ[x]/<x**3 + 1>

>>> QQ[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 $+ : M \times M \to M$ and $\times : R \times M \to M$ called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69] (page 1627). 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 $M, M^2, M^3 \ldots$, 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[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 is a subset of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ[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[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[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[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[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[x].free_module(0).is_zero()
True
>>> QQ[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[x].ideal(x)
>>> F = QQ[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[x].free_module(2)
```

```
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more conicisely, using the overloaded division operator:

```
>>> QQ[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 represantition.

Mostly called implicitly.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ[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[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[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[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[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[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[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[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 $\{f \in R | fN \subset M\}$.

```
>>> from sympy import QQ
>>> from sympy.abc import x, y
>>> F = QQ[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 prinicipal. 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[x].ideal(x**2)
>>> M = QQ[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[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[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 $f_1, \ldots, f_n$ over the ring R. Consider the homomorphism $\phi : R^n \to M$, given by sending $(r_1, \ldots, r_n) \to r_1 f_1 + \cdots + r_n f_n$. 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[x].free_module(2).submodule([1, 0], [1, 1]).syzygy_module().is_zero()
True
```

A slightly more interesting example:

```
>>> M = QQ[x, y].free_module(2).submodule([x, 2*x], [y, 2*y])
>>> S = QQ[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[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[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[x].ideal(x+1, x-1).contains(3)
True
>>> QQ[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[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[x, y].ideal(x).product(QQ[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 $I : J = \{x \in R | xJ \subset I\}$.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ[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 $I : J^\infty = \{x \in R | xJ^n \subset I$ for some $n\}$.

subset(*other*)

Returns True if `other` is is a subset of `self`.

Here `other` may be an ideal.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ[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[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[x].ideal(x**2 - 1).union(QQ[x].ideal((x+1)**2)) == QQ[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[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[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[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[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[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[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[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
>>> F = QQ[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[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 $f : M \to N$ satisfying various obvious properties (see [Atiyah69] (page 1627)) 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* $im(f)$. 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 $im(f) = N$. A bijective homomorphism is called an *isomorphism*. Equivalently, $ker(f) = 0$ and $im(f) = N$. (A related notion, which currently has no special name in the AGCA module, is that of the *cokernel*, $coker(f) = N/im(f)$.)

Suppose now M is an A-module. M is called *finitely generated* if there exists a surjective homomorphism $A^n \to M$ 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 $f : M \to N$ is an arbitrary morphism of A-modules. Then if K is a submodule of M, f naturally defines a new homomorphism $g : K \to N$ (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 $g : M/K \to N$ (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 $g : M \to N/L$, we say that g *factors* through f. Finally, if now L contains the image of f, then there is a natural homomorphism $g : M \to L$ (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 $A^n \to A^m$ 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, homomorphism
>>> from sympy.abc import x
>>> R = QQ[x]
>>> T = R.free_module(2)
```

If `domain` is a free module generated by $e_1, \ldots, e_n$, then `matrix` should be an n-element iterable $(b_1, \ldots, b_n)$ 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, them `matrix` determines a homomoprhism 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 homomoprhisms. Do not instantiate.

Instead, use the `homomorphism` function:

```
>>> from sympy import QQ, homomorphism
>>> from sympy.abc import x
>>> F = QQ[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 - cachd 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 $\phi : M \to N$, then compute $im(\phi) = \{\phi(x)|x \in M\}$. This is a submodule of N.

```
>>> from sympy import homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[x].free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> 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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 $\phi : M \to N$, then compute $ker(\phi) = \{x \in M | \phi(x) = 0\}$. This is a submodule of M.

```
>>> from sympy import homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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 homomorphism, QQ
>>> from sympy.abc import x
>>> F = QQ[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_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. $K(\alpha, \ldots)$.

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*, ***args*)
Returns numerical approximation of a.

exquo(*a*, *b*)
Exact quotient of a and b, implies something.

frac_field(**gens*)
Returns a fraction field, i.e. K(X).

from_AlgebraicField(*K1*, *a*, *K0*)
Convert an algebraic number to dtype.

from_ExpressionDomain(*K1*, *a*, *K0*)
Convert a EX object to dtype.

from_FF_gmpy(*K1*, *a*, *K0*)
Convert ModularInteger(mpz) to dtype.

from_FF_python(*K1*, *a*, *K0*)
Convert ModularInteger(int) to dtype.

from_FractionField(*K1*, *a*, *K0*)
Convert a rational function to dtype.

from_GlobalPolynomialRing(*K1*, *a*, *K0*)
Convert a polynomial to dtype.

from_PolynomialRing(*K1*, *a*, *K0*)
Convert a polynomial to dtype.

from_QQ_gmpy(*K1*, *a*, *K0*)
Convert a GMPY `mpq` object to `dtype`.

from_QQ_python(*K1*, *a*, *K0*)
Convert a Python `Fraction` object to `dtype`.

from_RR_mpmath(*K1*, *a*, *K0*)
Convert a mpmath `mpf` object to `dtype`.

from_ZZ_gmpy(*K1*, *a*, *K0*)
Convert a GMPY `mpz` object to `dtype`.

from_ZZ_python(*K1*, *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(**gens*)
Inject generators into this domain.

invert(*a*, *b*)
Returns inversion of `a mod 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*, ***args*)
Returns numerical approximation of `a`.

neg(*a*)
Returns `a` negated, implies `__neg__`.

numer(*a*)
Returns numerator of `a`.

of_type(*a*)
Check if `a` is of type `dtype`.

poly_ring(**gens*, ***opts*)
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(*K0*, *K1*, *gens=None*)
Returns a maximal domain containing K_0 and K_1.

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[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[x].ideal(x**2)
<x**2>
```

invert(*a*, *b*)
Returns inversion of a mod 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[x].quotient_ring(QQ[x].ideal(x**2))
QQ[x]/<x**2>
>>> QQ[x].quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ[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.simpledomain.**SimpleDomain**
Base class for simple domains, e.g. ZZ, 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. ZZ[x], ZZ(X).

Attributes

alias	
dtype	
one	
rep	
zero	

inject(**gens*)
Inject generators into this domain.

Concrete Domains

class sympy.polys.domains.**FiniteField**(*mod*, *symmetric=True*)
General class for finite fields.

Attributes

alias	
dom	
dtype	
mod	
one	
zero	

characteristic()
Return the characteristic of this domain.

from_FF_gmpy(*K1*, *a*, *K0=None*)
Convert `ModularInteger(mpz)` to `dtype`.

from_FF_python(*K1*, *a*, *K0=None*)
Convert `ModularInteger(int)` to `dtype`.

from_QQ_gmpy(*K1*, *a*, *K0=None*)
Convert GMPY's `mpq` to `dtype`.

from_QQ_python(*K1*, *a*, *K0=None*)
Convert Python's `Fraction` to `dtype`.

from_RR_mpmath(*K1*, *a*, *K0*)
Convert mpmath's `mpf` to `dtype`.

from_ZZ_gmpy(*K1*, *a*, *K0=None*)
Convert GMPY's `mpz` to `dtype`.

from_ZZ_python(*K1*, *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(*K1*, *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**
Create a generalized multivariate polynomial ring.

A generalized polynomial ring is defined by a ground field K, a set of generators (typically $x_1, \dots, x_n$) and a monomial order $<$. The monomial order can be global, local or mixed. In any case it induces a total ordering on the monomials, and there exists for every (non-zero) polynomial $f \in K[x_1, \dots, x_n]$ a well-defined "leading monomial" $LM(f) = LM(f, >)$. One can then define a multiplicative subset $S = S_> = \{f \in K[x_1, \dots, x_n] | LM(f) = 1\}$. The generalized polynomial ring corresponding to the monomial order is $R = S^{-1}K[x_1, \dots, x_n]$.

If $>$ is a so-called global order, that is 1 is the smallest monomial, then we just have $S = K$ and $R = K[x_1, \dots, x_n]$.

Examples

A few examples may make this clearer.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
```

Our first ring uses global lexicographic order.

```
>>> R1 = QQ.poly_ring(x, y, order=(("lex", x, y),))
```

The second ring uses local lexicographic order. Note that when using a single (non-product) order, you can just specify the name and omit the variables:

```
>>> R2 = QQ.poly_ring(x, y, order="ilex")
```

The third and fourth rings use a mixed orders:

```
>>> o1 = (("ilex", x), ("lex", y))
>>> o2 = (("lex", x), ("ilex", y))
>>> R3 = QQ.poly_ring(x, y, order=o1)
>>> R4 = QQ.poly_ring(x, y, order=o2)
```

We will investigate what elements of $K(x, y)$ are contained in the various rings.

```
>>> L = [x, 1/x, y/(1 + x), 1/(1 + y), 1/(1 + x*y)]
>>> test = lambda R: [f in R for f in L]
```

The first ring is just $K[x, y]$:

```
>>> test(R1)
[True, False, False, False, False]
```

The second ring is R1 localised at the maximal ideal (x, y):

```
>>> test(R2)
[True, False, True, True, True]
```

The third ring is R1 localised at the prime ideal (x):

```
>>> test(R3)
[True, False, True, False, True]
```

Finally the fourth ring is R1 localised at $S = K[x, y] \setminus yK[y]$:

```
>>> test(R4)
[True, False, False, True, False]
```

class sympy.polys.domains.**RationalField**

General class for rational fields.

Attributes

alias	
dtype	
one	
zero	

algebraic_field(**extension*)

Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \ldots)$.

from_AlgebraicField(*K1, 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 `ANP`

from_QQ_gmpy(*K1*, *a*, *K0*)
Convert a GMPY `mpq` object to `dtype`.

from_QQ_python(*K1*, *a*, *K0*)
Convert a Python `Fraction` object to `dtype`.

from_RR_mpmath(*K1*, *a*, *K0*)
Convert a mpmath `mpf` object to `dtype`.

from_ZZ_gmpy(*K1*, *a*, *K0*)
Convert a GMPY `mpz` object to `dtype`.

from_ZZ_python(*K1*, *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**(*dom*, **gens*)
A class for representing rational function fields.

Attributes

alias	
one	
rep	
zero	

denom(*a*)
Returns denominator of a.

dtype
alias of DMF

factorial(*a*)
Returns factorial of a.

frac_field(**gens*)
Returns a fraction field, i.e. $K(X)$.

from_FractionField(*K1*, *a*, *K0*)
Convert a fraction field element to another fraction field.

Examples

```
>>> from sympy.polys.polyclasses import DMF
>>> from sympy.polys.domains import ZZ, QQ
>>> from sympy.abc import x

>>> f = DMF(([ZZ(1), ZZ(2)], [ZZ(1), ZZ(1)]), ZZ)

>>> QQx = QQ.frac_field(x)
>>> ZZx = ZZ.frac_field(x)

>>> QQx.from_FractionField(f, ZZx)
(x + 2)/(x + 1)
```

from_GlobalPolynomialRing(*K1*, *a*, *K0*)
Convert a DMF object to dtype.

from_QQ_gmpy(*K1*, *a*, *K0*)
Convert a GMPY mpq object to dtype.

from_QQ_python(*K1*, *a*, *K0*)
Convert a Python Fraction object to dtype.

from_RR_mpmath(*K1*, *a*, *K0*)
Convert a mpmath mpf object to dtype.

from_ZZ_gmpy(*K1*, *a*, *K0*)
Convert a GMPY mpz object to dtype.

from_ZZ_python(*K1*, *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.

poly_ring(**gens*)
Returns a polynomial ring, i.e. $K[X]$.

to_sympy(*a*)
Convert a to a SymPy object.

class sympy.polys.domains.**ExpressionDomain**
A class for arbitrary expressions.

Attributes

alias	

class Expression(*ex*)
An arbitrary expression.

ExpressionDomain.**denom**(*a*)
Returns denominator of a.

ExpressionDomain.**dtype**
alias of Expression (page 1042)

ExpressionDomain.**from_ExpressionDomain**(*K1*, *a*, *K0*)
Convert a EX object to dtype.

ExpressionDomain.**from_FractionField**(*K1*, *a*, *K0*)
Convert a DMF object to dtype.

ExpressionDomain.**from_GlobalPolynomialRing**(*K1*, *a*, *K0*)
Convert a DMP object to dtype.

ExpressionDomain.**from_QQ_gmpy**(*K1*, *a*, *K0*)
Convert a GMPY mpq object to dtype.

ExpressionDomain.**from_QQ_python**(*K1*, *a*, *K0*)
Convert a Python Fraction object to dtype.

ExpressionDomain.**from_RR_mpmath**(*K1*, *a*, *K0*)
Convert a mpmath mpf object to dtype.

ExpressionDomain.**from_ZZ_gmpy**(*K1*, *a*, *K0*)
Convert a GMPY mpz object to dtype.

ExpressionDomain.**from_ZZ_python**(*K1*, *a*, *K0*)
Convert a Python int object to dtype.

ExpressionDomain.**from_sympy**(*a*)
Convert SymPy's expression to dtype.

ExpressionDomain.**get_field**()
Returns a field associated with `self`.

ExpressionDomain.**get_ring**()
Returns a ring associated with `self`.

ExpressionDomain.**is_negative**(*a*)
Returns True if `a` is negative.

ExpressionDomain.**is_nonnegative**(*a*)
Returns True if `a` is non-negative.

ExpressionDomain.**is_nonpositive**(*a*)
Returns True if `a` is non-positive.

ExpressionDomain.**is_positive**(*a*)
Returns True if `a` is positive.

ExpressionDomain.**numer**(*a*)
Returns numerator of `a`.

ExpressionDomain.**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

dtype	
mod	
one	
zero	

class sympy.polys.domains.**GMPYFiniteField**(*mod, symmetric=True*)
Finite field based on Python's integers.

Attributes

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(*f*)

Returns the leading coefficient of `f`.

TC(*f*)

Returns the trailing coefficient of `f`.

abs(*f*)

Make all coefficients in `f` positive.

add(*f*, *g*)

Add two multivariate polynomials `f` and `g`.

add_ground(*f*, *c*)

Add an element of the ground domain to `f`.

all_coeffs(*f*)

Returns all coefficients from `f`.

all_monoms(*f*)

Returns all monomials from `f`.

all_terms(*f*)

Returns all terms from a `f`.

cancel(*f*, *g*, *include=True*)

Cancel common factors in a rational function `f/g`.

clear_denoms(*f*)

Clear denominators, but keep the ground domain.

coeffs(*f*, *order=None*)

Returns all non-zero coefficients from `f` in lex order.

cofactors(*f*, *g*)

Returns GCD of `f` and `g` and their cofactors.

compose(*f*, *g*)

Computes functional composition of `f` and `g`.

content(*f*)

Returns GCD of polynomial coefficients.

convert(*f*, *dom*)

Convert the ground domain of `f`.

count_complex_roots(*f*, *inf=None*, *sup=None*)

Return the number of complex roots of `f` in `[inf, sup]`.

count_real_roots(*f*, *inf=None*, *sup=None*)

Return the number of real roots of `f` in `[inf, sup]`.

decompose(*f*)

Computes functional decomposition of `f`.

deflate(*f*)

Reduce degree of f by mapping x_i^m to y_i.

degree(*f*, *j=0*)

Returns the leading degree of `f` in `x_j`.

degree_list(*f*)
Returns a list of degrees of `f`.

diff(*f, m=1, j=0*)
Computes the `m`-th order derivative of `f` in `x_j`.

discriminant(*f*)
Computes discriminant of `f`.

div(*f, g*)
Polynomial division with remainder of `f` and `g`.

eject(*f, dom, front=False*)
Eject selected generators into the ground domain.

eval(*f, a, j=0*)
Evaluates `f` at the given point `a` in `x_j`.

exclude(*f*)
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(*f, g*)
Computes polynomial exact quotient of `f` and `g`.

exquo_ground(*f, c*)
Exact quotient of `f` by a an element of the ground domain.

factor_list(*f*)
Returns a list of irreducible factors of `f`.

factor_list_include(*f*)
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(*f, g*)
Returns polynomial GCD of `f` and `g`.

gcdex(*f, g*)
Extended Euclidean algorithm, if univariate.

gff_list(*f*)
Computes greatest factorial factorization of `f`.

half_gcdex(*f, g*)
Half extended Euclidean algorithm, if univariate.

homogeneous_order(*f*)
Returns the homogeneous order of `f`.

inject(*f, front=False*)
Inject ground domain generators into `f`.

integrate(*f, m=1, j=0*)
Computes the `m`-th order indefinite integral of `f` in `x_j`.

intervals(*f, all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)
Compute isolating intervals for roots of `f`.

invert(*f, 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(*f*)
Returns l1 norm of `f`.

lcm(*f, g*)
Returns polynomial LCM of `f` and `g`.

lift(*f*)
Convert algebraic coefficients to rationals.

max_norm(*f*)
Returns maximum norm of `f`.

monic(*f*)
Divides all coefficients by `LC(f)`.

monoms(*f*, *order=None*)
Returns all non-zero monomials from f in lex order.

mul(*f*, *g*)
Multiply two multivariate polynomials f and g.

mul_ground(*f*, *c*)
Multiply f by a an element of the ground domain.

neg(*f*)
Negate all cefficients in f.

nth(*f*, **N*)
Returns the n-th coefficient of f.

pdiv(*f*, *g*)
Polynomial pseudo-division of f and g.

per(*f*, *rep*, *dom=None*, *kill=False*, *ring=None*)
Create a DMP out of the given representation.

permute(*f*, *P*)
Returns a polynomial in $K[x_{P(1)}, ..., x_{P(n)}]$.

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(*f*, *g*)
Polynomial exact pseudo-quotient of f and g.

pow(*f*, *n*)
Raise f to a non-negative power n.

pquo(*f*, *g*)
Polynomial pseudo-quotient of f and g.

prem(*f*, *g*)
Polynomial pseudo-remainder of f and g.

primitive(*f*)
Returns content and a primitive form of f.

quo(*f*, *g*)
Computes polynomial quotient of f and g.

quo_ground(*f*, *c*)
Quotient of f by a an element of the ground domain.

refine_root(*f*, *s*, *t*, *eps=None*, *steps=None*, *fast=False*)
Refine an isolating interval to the given precision.

eps should be a rational number.

rem(*f*, *g*)
Computes polynomial remainder of `f` and `g`.

resultant(*f*, *g*, *includePRS=False*)
Computes resultant of `f` and `g` via PRS.

revert(*f*, *n*)
Compute `f**(-1)` mod `x**n`.

shift(*f*, *a*)
Efficiently compute Taylor shift `f(x + a)`.

slice(*f*, *m*, *n*, *j=0*)
Take a continuous subsequence of terms of `f`.

sqf_list(*f*, *all=False*)
Returns a list of square-free factors of `f`.

sqf_list_include(*f*, *all=False*)
Returns a list of square-free factors of `f`.

sqf_norm(*f*)
Computes square-free norm of `f`.

sqf_part(*f*)
Computes square-free part of `f`.

sqr(*f*)
Square a multivariate polynomial `f`.

sturm(*f*)
Computes the Sturm sequence of `f`.

sub(*f*, *g*)
Subtract two multivariate polynomials `f` and `g`.

sub_ground(*f*, *c*)
Subtract an element of the ground domain from `f`.

subresultants(*f*, *g*)
Computes subresultant PRS sequence of `f` and `g`.

terms(*f*, *order=None*)
Returns all non-zero terms from `f` in lex order.

terms_gcd(*f*)
Remove GCD of terms from the polynomial `f`.

to_dict(*f*, *zero=False*)
Convert `f` to a dict representation with native coefficients.

to_exact(*f*)
Make the ground domain exact.

to_field(*f*)
Make the ground domain a field.

to_ring(*f*)
Make the ground domain a ring.

to_sympy_dict(*f*, *zero=False*)
Convert `f` to a dict representation with SymPy coefficients.

to_tuple(*f*)
Convert `f` to a tuple representation with native coefficients.

This is needed for hashing.

total_degree(*f*)
Returns the total degree of `f`.

trunc(*f*, *p*)
Reduce `f` modulo a constant `p`.

unify(*f*, *g*)
Unify representations of two multivariate polynomials.

class `sympy.polys.polyclasses.`**DMF**(*rep*, *dom*, *lev=None*, *ring=None*)
Dense Multivariate Fractions over K.

add(*f*, *g*)
Add two multivariate fractions `f` and `g`.

cancel(*f*)
Remove common factors from `f.num` and `f.den`.

denom(*f*)
Returns the denominator of `f`.

exquo(*f*, *g*)
Computes quotient of fractions `f` and `g`.

frac_unify(*f*, *g*)
Unify representations of two multivariate fractions.

half_per(*f*, *rep*, *kill=False*)
Create a DMP out of the given representation.

invert(*f*, *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(*f*, *g*)
Multiply two multivariate fractions `f` and `g`.

neg(*f*)
Negate all cefficients in `f`.

numer(*f*)
Returns the numerator of `f`.

per(*f*, *num*, *den*, *cancel=True*, *kill=False*, *ring=None*)
Create a DMF out of the given representation.

poly_unify(*f*, *g*)
Unify a multivariate fraction and a polynomial.

pow(*f*, *n*)
Raise `f` to a non-negative power `n`.

quo(*f*, *g*)
Computes quotient of fractions `f` and `g`.

sub(*f*, *g*)
Subtract two multivariate fractions `f` and `g`.

class `sympy.polys.polyclasses.`**ANP**(*rep*, *mod*, *dom*)
Dense Algebraic Number Polynomials over a field.

LC(*f*)
Returns the leading coefficient of `f`.

TC(*f*)
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(*f*, *n*)
Raise `f` to a non-negative power `n`.

to_dict(*f*)
Convert `f` to a dict representation with native coefficients.

to_list(*f*)
Convert `f` to a list representation with native coefficients.

to_sympy_dict(*f*)
Convert `f` to a dict representation with SymPy coefficients.

to_sympy_list(*f*)
Convert `f` to a list representation with SymPy coefficients.

to_tuple(*f*)
Convert `f` to a tuple representation with native coefficients.

This is needed for hashing.

unify(*f*, *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 $K[X_0, \ldots, X_u]$. 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 ... 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_0` in `K[X]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree

>>> dmp_degree([[[]]], 2)
-1

>>> 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 * 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.domains import ZZ
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.densebasic import dmp_convert

>>> f = [[DMP([1], ZZ)], [DMP([2], ZZ)]]
>>> g = [[ZZ(1)], [ZZ(2)]]

>>> dmp_convert(f, 1, ZZ['x'], ZZ)
[[1], [2]]

>>> dmp_convert(g, 1, ZZ, ZZ['x'])
[[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([[[[[1]]]]], 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[..x_i..x_j..] to K[..x_j..x_i..].

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]], [[]]]
>>> 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)},..,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.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inject

>>> K = ZZ['x', 'y']

>>> dmp_inject([K([[1]]), K([[1], [2]])], 0, K)
([[[1]], [[1], [2]]], 2)
>>> dmp_inject([K([[1]]), K([[1], [2]])], 0, K, 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

>>> K = ZZ['x', 'y']

>>> dmp_eject([[[1]], [[1], [2]]], 2, K)
[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.densetools.**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.densetools.**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.densetools.**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.densetools.**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.densetools.**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.densetools.**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.densetools.**dmp_eval_tail**(*f, A, u, K*)

Evaluate a polynomial at `x_j = a_j, ...` 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.densetools.**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.densetools.**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.densetools.**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.densetools.**dup_monic**(*f, K*)

Divide all coefficients by 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.densetools.**dmp_ground_monic**(*f, u, K*)

Divide all coefficients by 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.densetools.**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.densetools.**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.densetools.**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.densetools.**dup_real_imag**(*f*, *K*)

Return bivariate polynomials `f1` and `f2`, such that `f = f1 + f2*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.densetools.**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.densetools.**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, ..., f_n]`, where:

```
f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n))
```

and `f_2, ..., 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 o g = f(x + b) o (g - b)`
2. `x**n o x**m = x**m o x**n`
3. `T_n o T_m = T_m o T_n`

where `T_n` and `T_m` are Chebyshev polynomials.

References

1. [Kozen89] (page 1626)

[Kozen89] (page 1626)

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.densetools.**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)` mod `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,...,u_n` and a set of co-prime integer moduli `m_0,...,m_n`, returns an integer `u`, such that `u = u_i mod m_i` for `i = ‘‘0,...,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 275) a higher level crt routine

sympy.ntheory.modular.solve_congruence (page 277)

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 mod 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 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 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 GF(p)[x] polynomial from 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 GF(p)[x] polynomial to 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 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 GF(p)[x] and a in 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 GF(p)[x] and a in 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 `GF(p)[x]` and `a` in `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 `GF(p)[x]` and `a` in `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 `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 `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 `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 `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*h` where `f`, `g`, `h` in `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*h` where `f`, `g`, `h` in `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 `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 `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 = q*g + r`.

Consider polynomials `x**3 + x + 1` and `x**2 + x` in 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 1626)
2. [Gathen99] (page 1626)

[Monagan93] (page 1626), [Gathen99] (page 1626)

sympy.polys.galoistools.**gf_rem**(*f, g, p, K*)

Compute polynomial remainder in `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 `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 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 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 `GF(p)[x]/(g)` using repeated squaring.

Given polynomials `f` and `g` in `GF(p)[x]` and a non-negative integer `n`, efficiently computes `f**n (mod g)` i.e. the remainder of `f**n` from division by `g`, using the repeated squaring algorithm.

References

1. [Gathen99] (page 1626)

[Gathen99] (page 1626)

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 `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 `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 `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 `GF(p)[x]`.

Given polynomials `f` and `g` in `GF(p)[x]`, computes polynomials `s`, `t` and `h`, such that `h = 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 `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 = 5*x + 6` and `t = 6`, and additionally `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 1626)

[Gathen99] (page 1626)

sympy.polys.galoistools.**gf_monic**(*f, p, K*)

Compute LC and a monic polynomial in `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 `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 `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, ..., 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 `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 `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 `GF(p)[x]/(f)`.

Given a polynomial `f` in `GF(p)[x]`, polynomials `a`, `b`, `c` in the quotient ring `GF(p)[x]/(f)` such that `b = c**t (mod 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 mod f` and `c = x mod 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 1626)

[Gathen92] (page 1626)

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 `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 `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 `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 `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 `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 `GF(p)[x]` polynomial.

Given a polynomial `f` in `GF(p)[x]`, returns the leading coefficient of `f` and a square-free decomposition `f_1**e_1 f_2**e_2 ... f_k**e_k` such that all `f_i` are monic polynomials and `(f_i, f_j)` for `i != j` are co-prime and `e_1 ... 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 `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 1626)

[Geddes92] (page 1626)

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 `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 `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 `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 `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 `GF(p)[x]`.

Given a possibly non square-free polynomial `f` in `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 `gcd(f_i, f_j) == 1`, for `i != 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 = (7*x + 1) (x + 2)**2` in `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 `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 1626)

[Gathen99] (page 1626)

`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) congruent 0 mod(n).

n is divided into canonical factors and f(x) cong 0 mod(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] congruent 0 mod(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.

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 $f_1, f_2, \ldots$.

`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 $B = x^2 y f_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

$A = xyf_5$ divides $B = x^2 y f_5$

```
>>> 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
```

$A = xy^2 f_5$ 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 coeffcient 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:

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)
```

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)
```

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]
>>> beta = [[-1], [1], [9, 0, 0, 0, 0, 0, 0, 0, 0]]
>>> delta = [0, 1, 1]

>>> R.dmp_inner_subresultants(f, g) == (prs, beta, delta)
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 $Z[X]$.

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 $Q[X]$.

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 = quo(f, h)`, and `cfg = 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 = quo(f, h)`, and `cfg = 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 $Z[X]$.

Given univariate polynomials f and g in $Z[X]$, 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 proces 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 1626)

[Liao95] (page 1626)

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 = quo(f, h)`, and `cfg = 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 = quo(f, h)`, and `cfg = 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 1626)

[Gathen99] (page 1626)

`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, ..., 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 1626)

[Gathen99] (page 1626)

`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 polnomial.

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 polnomial.

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 $n >= 1$. Otherwise returns None.

Factorization is performed using using cyclotomic decomposition of f, which makes this method much faster that any other direct factorization approach (e.g. Zassenhaus's).

References

1. [Weisstein09] (page 1626)

[Weisstein09] (page 1626)

sympy.polys.factortools.**dup_zz_factor_sqf**(f, K)

Factor square-free (non-primitive) polyomials 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 $f_1, ..., f_n$ 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 1626)

[Gathen99] (page 1626)

`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 $Z[x_1, ..., x_n]$, 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 $Z[x_2, ..., x_n][x_1]$, for which an evaluation mapping is computed:

```
x_2 -> a_2, ..., x_n -> a_n
```

where a_i, for $i = 2, ..., n$, 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 1626)
2. [Geddes92] (page 1626)

[Wang78] (page 1626), [Geddes92] (page 1626)

`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 $f_1, ..., f_n$ 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 1626)

[Gathen99] (page 1626)

`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 in 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*)
Compute LCM(LM(p1), LM(p2))/LM(p1)*p1 - LCM(LM(p1), LM(p2))/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 $R = K[x_1, \ldots, x_n]$, 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 $in(S) = in(N)$, where for any subset X of F, $in(X)$ 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

Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

5.17 Printing System

See the *Printing* (page 10) section in Tutorial for introduction into printing.

This guide documents the printing system in SymPy and how it works internally.

5.17.1 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.

Some more information how the single concepts work and who should use which:

1. The object prints itself

   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. Therefor 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.

   Nevertheless, to get a fitting representation, the printers look for a specific method in every object, that will be called if it's available and is then responsible for the representation. The name of that method depends on the specific printer and is defined under Printer.printmethod.

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(...), in order to dispatch, we will try calling printer methods 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, we proceed with trying Rational bases in the inheritance order.

3. As 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).

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.

Basically, if you want to implement a printer, all you have to do is:

1. Subclass Printer.
2. Define Printer.printmethod in your subclass. If a object has a method with that name, this method will be used for printing.
3. In your subclass, define `_print_<CLASS>` methods

   For each class you want to provide printing to, define an appropriate method how to do it. For example if you want a class FOO to be printed in its own way, define _print_FOO:

   ```
   def _print_FOO(self, e):
       ...
   ```

   this should return how FOO instance e is printed

   Also, if `BAR` is a subclass of `FOO`, `_print_FOO(bar)` will be called for instance of `BAR`, if no `_print_BAR` is provided. Thus, usually, we don't need to provide printing routines for every class we want to support – only generic routine has to be provided for a set of classes.

   A good example for this are functions - for example `PrettyPrinter` only defines `_print_Function`, and there is no `_print_sin`, `_print_tan`, etc...

   On the other hand, a good printer will probably have to define separate routines for `Symbol`, `Atom`, `Number`, `Integral`, `Limit`, etc...
4. If convenient, override `self.emptyPrinter`

   This callable will be called to obtain printing result as a last resort, that is when no appropriate print method was found for an expression.

Examples of overloading StrPrinter:

```
from sympy import Basic, Function, Symbol
from sympy.printing.str import StrPrinter

class CustomStrPrinter(StrPrinter):
    """
    Examples of how to customize the StrPrinter for both a SymPy class and a
    user defined class subclassed from the SymPy Basic class.
    """

    def _print_Derivative(self, expr):
        """
        Custom printing of the SymPy Derivative class.

        Instead of:

        D(x(t), t) or D(x(t), t, t)
```

```
        We will print:

        x'     or     x''

        In this example, expr.args == (x(t), t), and expr.args[0] == x(t), and
        expr.args[0].func == x
        """
        return str(expr.args[0].func) + "''"*len(expr.args[1:])

    def _print_MyClass(self, expr):
        """
        Print the characters of MyClass.s alternatively lower case and upper
        case
        """
        s = ""
        i = 0
        for char in expr.s:
            if i % 2 == 0:
                s += char.lower()
            else:
                s += char.upper()
            i += 1
        return s

# Override the __str__ method of to use CustromStrPrinter
Basic.__str__ = lambda self: CustomStrPrinter().doprint(self)
# Demonstration of CustomStrPrinter:
t = Symbol('t')
x = Function('x')(t)
dxdt = x.diff(t)            # dxdt is a Derivative instance
d2xdt2 = dxdt.diff(t)       # dxdt2 is a Derivative instance
ex = MyClass('I like both lowercase and upper case')

print dxdt
print d2xdt2
print ex
```

The output of the above code is:

```
x'
x''
i lIkE BoTh lOwErCaSe aNd uPpEr cAsE
```

By overriding Basic.__str__, we can customize the printing of anything that is subclassed from Basic.

Attributes

printmethod	

printmethod = None

doprint(*expr*)

Returns printer's representation for expr (as a string)

_print(*expr*, **args*, ***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.

classmethod set_global_settings(***settings*)
Set system-wide printing settings.

5.17.2 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*, ***settings*)
Prints expr in pretty form.

pprint is just a shortcut for this function.

Parameters **expr** : expression

the expression to print

wrap_line : bool, optional

line wrapping enabled/disabled, defaults to True

num_columns : bool, optional

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

use unicode characters, such as the Greek letter pi instead of the string pi.

full_prec : bool or string, optional

use full precision. Default to "auto"

order : bool or string, optional

set to 'none' for long expressions if slow; default is None

5.17.3 CCodePrinter

This class implements C code printing (i.e. it converts Python expressions to strings of C code).

Usage:

```
>>> from sympy.printing import print_ccode
>>> from sympy.functions import sin, cos, Abs
>>> from sympy.abc import x
>>> print_ccode(sin(x)**2 + cos(x)**2)
pow(sin(x), 2) + pow(cos(x), 2)
>>> print_ccode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_ccode(Abs(x**2))
fabs(pow(x, 2))
```

sympy.printing.ccode.**known_functions = {'ceiling': [(<function <lambda> at 0x00000000043689**

class sympy.printing.ccode.**CCodePrinter**(*settings={}*)

A printer to convert python expressions to strings of c code

printmethod = '_ccode'

doprint(*expr, assign_to=None*)

Actually format the expression as C code.

indent_code(*code*)

Accepts a string of code or a list of code lines

sympy.printing.ccode.**ccode**(*expr, assign_to=None, **settings*)

Converts an expr to a string of c code

Parameters **expr** : sympy.core.Expr

a sympy expression to be converted

precision : optional

the precision for numbers such as pi [default=15]

user_functions : 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 : 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 more programmer-friendly data structure.

Examples

```
>>> from sympy import ccode, symbols, Rational, sin, ceiling, Abs
>>> x, tau = symbols(["x", "tau"])
>>> ccode((2*tau)**Rational(7,2))
'8*sqrt(2)*pow(tau, 7.0L/2.0L)'
>>> ccode(sin(x), assign_to="s")
's = sin(x);'
```

```
>>> custom_functions = {
...   "ceiling": "CEIL",
...   "Abs": [(lambda x: not x.is_integer, "fabs"),
...           (lambda x: x.is_integer, "ABS")]
... }
>>> ccode(Abs(x) + ceiling(x), user_functions=custom_functions)
'fabs(x) + CEIL(x)'
```

`sympy.printing.ccode.print_ccode(expr, **settings)`

Prints C representation of the given expression.

5.17.4 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, **settings)`

Converts an expr to a string of Fortran 77 code

Parameters **expr** : sympy.core.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 Fortran expression is assigned. (This is helpful in case of line-wrapping.)

precision : optional

the precision for numbers such as pi [default=15]

user_functions : optional

A dictionary where keys are FunctionClass instances and values are there string representations.

human : optional

If True, the result is a single string that may contain some parameter statements for the number symbols. If False, the same information is returned in a more programmer-friendly data structure.

source_format : optional

The source format can be either 'fixed' or 'free'. [default='fixed']

Examples

```
>>> from sympy import fcode, symbols, Rational, pi, sin
>>> 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)'
>>> print fcode(pi)
```

```
parameter (pi = 3.14159265358979d0)
pi
```

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=None*)

A printer to convert sympy expressions to strings of Fortran code

printmethod = '_fcode'

doprint(*expr*)

Returns Fortran code for expr (as a string)

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 only top-level piecewise functions are supported due to the lack of a conditional operator in Fortran. Nested piecewise functions would require the introduction of temporary variables, which is a type of expression manipulation that goes beyond the scope of `fcode`.

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 = 2.71828182845905d0)
      parameter (pi = 3.14159265358979d0)
      x - pi**2 - E
>>> print(fcode(x - pi**2 - E, precision=25))
      parameter (E = 2.718281828459045235360287d0)
      parameter (pi = 3.141592653589793238462643d0)
      x - pi**2 - E
>>> 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 Fortran:
C     gamma(x)
      -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 can not be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```
>>> fcode(1 - gamma(x)**2, human=False)
(set(), 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)
(set([(pi, '3.14159265358979d0')]), set(), '      x - pi**2')
```

5.17.5 Gtk

You can print to a grkmathview widget using the function `print_gtk` located in `sympy.printing.gtk` (it requires to have installed gtkmatmatview 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

5.17.6 LambdaPrinter

This classes implements printing to strings that can be used by the `sympy.utilities.lambdify.lambdify()` (page 1365) function.

class `sympy.printing.lambdarepr.`**`LambdaPrinter`**(*settings=None*)

This printer converts expressions into strings that can be used by lambdify.

`printmethod = '_sympystr'`

`sympy.printing.lambdarepr.`**`lambdarepr`**(*expr, **settings*)

Returns a string usable for lambdifying.

5.17.7 LatexPrinter

This class implements LaTeX printing. See `sympy.printing.latex`.

`sympy.printing.latex.`**`accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', 's`**

list() -> new empty list list(iterable) -> new list initialized from iterable's items

class `sympy.printing.latex.`**`LatexPrinter`**(*settings=None*)

`printmethod = '_latex'`

`sympy.printing.latex.`**`latex`**(*expr, **settings*)

Convert the given expression to LaTeX representation.

```
>>> from sympy import latex, pi, sin, asin, Integral, Matrix, Rational
>>> from sympy.abc import x, y, mu, r, tau

>>> print latex((2*tau)**Rational(7,2))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

order: Any of the supported monomial orderings (currently "lex", "grlex", or "grevlex"), "old", and "none". This parameter does nothing for Mul objects. Setting order to "old" uses the compatibility ordering for Add defined in Printer. For very large expressions, set the 'order' keyword to 'none' if speed is a concern.

mode: Specifies how the generated code will be delimited. 'mode' can be one of 'plain', 'inline', 'equation' or 'equation*'. If 'mode' is set to 'plain', then the resulting code will not be delimited at all (this is the default). If 'mode' is set to 'inline' then inline LaTeX $ $ will be used. If 'mode' is set to 'equation' or 'equation*', the resulting code will be enclosed in the 'equation' or 'equation*' environment (remember to import 'amsmath' for 'equation*'), unless the 'itex' option is set. In the latter case, the $$ $$ syntax is used.

```
>>> 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')
$8 \sqrt{2} \tau^{\frac{7}{2}}$

>>> 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. The default value is 2.

```
>>> 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", "bmatrix", etc. Defaults to "smallmatrix".

```
>>> print latex(Matrix(2, 1, [x, y]), mat_str = "array")
\left[\begin{array}{}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{smallmatrix}{}x\\y\end{smallmatrix}\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}
```

Besides all Basic based expressions, you can recursively convert Python containers (lists, tuples and dicts) and also SymPy matrices:

```
>>> print latex([2/x, y], mode='inline')
$\begin{bmatrix}2 / x, & y\end{bmatrix}$
```

`sympy.printing.latex.`**`print_latex`**(*expr*, ***settings*)

Prints LaTeX representation of the given expression.

5.17.8 MathMLPrinter

This class is responsible for MathML printing. See `sympy.printing.mathml`.

More info on mathml content: http://www.w3.org/TR/MathML2/chapter4.html

class `sympy.printing.mathml.`**`MathMLPrinter`**(*settings=None*)

Prints an expression to the MathML markup language

Whenever possible tries to use Content markup and not Presentation markup.

References: http://www.w3.org/TR/MathML2/

printmethod = '_mathml'

doprint(*expr*)

Prints the expression as MathML.

mathml_tag(*e*)

Returns the MathML tag for an expression.

sympy.printing.mathml.**mathml**(*expr, **settings*)
Returns the MathML representation of expr

sympy.printing.mathml.**print_mathml**(*expr, **settings*)
Prints a pretty representation of the MathML code for expr

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>
```

5.17.9 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)
```

5.17.10 ReprPrinter

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

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

5.17.11 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

5.17.12 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 an 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.abc import x
>>> print_tree(x**2)
Pow: x**2
+-Symbol: x
| comparable: False
+-Integer: 2
  real: True
  nonzero: True
  comparable: True
  commutative: True
  infinitesimal: False
  unbounded: False
  noninteger: False
```

```
zero: False
complex: True
bounded: True
rational: True
integer: True
imaginary: False
finite: True
irrational: False
```

See also: tree()

5.17.13 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, outputTexFile=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 genereted file to this filename instead.

There is also support for writing to a StringIO like object, which needs to be passed to the 'outputbuffer' argument.

```
>>> from StringIO import StringIO
>>> obj = StringIO()
>>> preview(x + y, output='png', viewer='StringIO',
...         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 ('dvioptions' 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")
```

5.17.14 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.**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 = {'And': 30, 'Add': 40, 'Pow': 60, 'Mul': 50, 'Not': 100, 'R**
Default precedence values for some basic types.

sympy.printing.precedence.**PRECEDENCE_VALUES = {'Sub': 40, 'factorial': 60, 'Add': 40, 'MatAdd': 4**
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 = {'FracElement': <function precedence_FracE**
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.

5.17.15 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', 'epsil**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

sympy.printing.pretty.pretty_symbology.**digit_2txt = {'1': 'ONE', '0': 'ZERO', '3': 'THREE', '2':**

sympy.printing.pretty.pretty_symbology.**symb_2txt = {'int': 'INTEGRAL', '{}': 'CURLY BRACKET**

The following functions return the Unicode subscript/superscript version of the character.

sympy.printing.pretty.pretty_symbology.**sub = {')': u'\u208e', 'chi': u'\u1d6a', '+': u'\u208a', '-'**

sympy.printing.pretty.pretty_symbology.**sup = {')': u'\u207e', 'i': u'\u2071', '(': u'\u207d', '+': u**

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: u'\u221a', 3: u'\u221b', 4: u'\u221c'}**

sympy.printing.pretty.pretty_symbology.**VF**(*txt*)

sympy.printing.pretty.pretty_symbology.**frac = {(1, 3): u'\u2153', (5, 6): u'\u215a', (1, 4): u'\xb**

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 = {'Integers': u'\u2124', 'NegativeInfinity'**

sympy.printing.pretty.pretty_symbology.**pretty_atom**(*atom_name, default=None*)
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_nougly=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.

5.17.16 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> at 0x000000001A5D0358>, maxdepth=None, repeat=True, labelfunc=<type '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"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol(x)_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#########
# Edges #
#########

"Add(Integer(2), Symbol(x))_()" -> "Integer(2)_(0,)";
"Add(Integer(2), Symbol(x))_()" -> "Symbol(x)_(1,)";
}
```

5.18 Plotting Module

5.18.1 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.

5.18.2 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(**args*)
Adds one more graph to the figure.

extend(*arg*)
Adds the series from another plot or a list of series.

5.18.3 Plotting Function Reference

sympy.plotting.plot.**plot**(**args, **kwargs*)
Plots a function of a single variable.

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 1132), 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 `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 1132), `Parametric2DLineSeries` (page 1143)

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 1132), `SurfaceOver2DRangeSeries` (page 1144)

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 1132), `Parametric3DLineSeries` (page 1144)

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 1132), `ParametricSurfaceSeries` (page 1144)

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
```

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, *args, **kwargs*)

A plot function to plot implicit equations / inequalities.

Arguments

- `expr` : The equation / inequality that is to be plotted.
- `(x, xmin, xmax)` optional, 3-tuple denoting the range of symbol `x`
- `(y, ymin, ymax)` optional, 3-tuple denoting the range of symbol `y`

The following arguments can be passed as named parameters.

- **`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

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.

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))
```

5.18.4 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*)

Representation for Implicit plot

5.19 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.

5.19.1 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.

5.19.2 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 is 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.

5.19.3 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]
```

5.19.4 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()
```

5.19.5 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) -> 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->red, y->green, and x->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.

5.19.6 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]
```

5.20 Assumptions module

5.20.1 Contents

Ask

Module for querying SymPy objects about assumptions.

class `sympy.assumptions.ask.`**`Q`**
Supported ask keys.

sympy.assumptions.ask.**ask**(*proposition, assumptions=True, context=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(x*y), Q.integer(x) &  Q.integer(y))
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 variables `known_facts` and `known_facts_keys` 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):
...         import math
...         return ask(Q.integer(math.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.

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

```
>>> Q.prime(7)
Q.prime(7)
```

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:

```
>>> Q.is_true(x > 1)
Q.is_true(x > 1)
>>> Q.is_true(S(1) < x)
Q.is_true(1 < x)
```

eval(*expr*, *assumptions=True*)

Evaluate self(expr) under the given assumptions.

This uses only direct resolution methods, not logical inference.

sympy.assumptions.assume.**assuming**(**args*, ***kwds*)

Context manager for assumptions

```
>>> from __future__ import with_statement
>>> 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_exp**(*expr, assumptions*)

Handler for exponential function.

```
>>> from sympy import Symbol, Q, exp, I, pi
>>> from sympy.assumptions.refine import refine_exp
>>> from sympy.abc import x
>>> refine_exp(exp(pi*I*2*x), Q.real(x))
>>> refine_exp(exp(pi*I*2*x), Q.integer(x))
1
```

Handlers

Contents

Calculus This module contains query handlers responsible for calculus queries: infinitesimal, bounded, etc.

class `sympy.assumptions.handlers.calculus.`**`AskBoundedHandler`**

Handler for key 'bounded'.

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 AskBoundedHandler
>>> from sympy.abc import x
>>> a = AskBoundedHandler()
>>> a.Symbol(x, Q.positive(x)) == None
True
>>> a.Symbol(x, Q.bounded(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()<=1 ** Positive -> Bounded Abs()>=1 ** Negative -> Bounded Otherwise unknown

static Symbol(*expr, assumptions*)
Handles Symbol.

Examples:

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.handlers.calculus import AskBoundedHandler
>>> from sympy.abc import x
>>> a = AskBoundedHandler()
>>> a.Symbol(x, Q.positive(x)) == None
True
>>> a.Symbol(x, Q.bounded(x))
True
```

class sympy.assumptions.handlers.calculus.**AskInfinitesimalHandler**
Handler for key 'infinitesimal' Test that a given expression is equivalent to an infinitesimal number

static Add(*expr, assumptions*)
Infinitesimal*Bounded -> Infinitesimal

static Mul(*expr, assumptions*)
Infinitesimal*Bounded -> Infinitesimal

static Pow(*expr, assumptions*)
Infinitesimal*Bounded -> Infinitesimal

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*Antihermitian -> !Hermitian Antihermitian*Antihermitian -> 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 numbers, 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**integer -> Imaginary if integer % 2 == 1 Imaginary**integer -> real if integer % 2 == 0 Imaginary**Imaginary -> ? Imaginary**Real -> ?

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 Integer*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 + !Rational -> ?

static Mul(*expr, assumptions*)
Rational + Rational -> Rational Rational + !Rational -> !Rational !Rational + !Rational -> ?

static Pow(*expr, assumptions*)
Rational ** Integer -> Rational Irrational ** Rational -> Irrational Rational ** Irrational -> ?

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 Real**Imaginary -> ? Imaginary**Real -> ? Real**Real -> ?

Queries are used to ask information about expressions. Main method for this is ask():

sympy.assumptions.ask.**ask**(*proposition*, *assumptions=True*, *context=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(x*y), Q.integer(x) &  Q.integer(y))
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.

5.20.2 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 a class in sympy.assumptions 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')
>>> facts = Q.positive(x), Q.positive(y)
>>> with assuming(*facts):
...     print(ask(Q.positive(2*x + y)))
True
```

5.20.3 Supported predicates

bounded

Test that a function is bounded with respect to its variables. For example, sin(x) is a bounded functions, but exp(x) is not.

Examples:

```
>>> from sympy import *
>>> x = Symbol('x')
>>> ask(Q.bounded(exp(x)), ~Q.bounded(x))
False
>>> ask(Q.bounded(exp(x)) , Q.bounded(x))
True
>>> ask(Q.bounded(sin(x)), ~Q.bounded(x))
True
```

commutative

Test that objects are commutative. By default, symbols in SymPy are considered commutative except otherwise stated.

Examples:

```
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> ask(Q.commutative(x))
True
>>> ask(Q.commutative(x), ~Q.commutative(x))
False
>>> ask(Q.commutative(x*y), ~Q.commutative(x))
False
```

complex

Test that expression belongs to the field of complex numbers.

Examples:

```
>>> from sympy import *
>>> ask(Q.complex(2))
True
>>> ask(Q.complex(I))
True
>>> x, y = symbols('x,y')
>>> ask(Q.complex(x+I*y), Q.real(x) & Q.real(y))
True
```

even

Test that expression represents an even number, that is, an number that can be written in the form 2*n, n integer.

Examples:

```
>>> from sympy import *
>>> ask(Q.even(2))
True
>>> n = Symbol('n')
>>> ask(Q.even(2*n), Q.integer(n))
True
```

extended_real

Test that an expression belongs to the field of extended real numbers, that is, real numbers union {Infinity, -Infinity}.

Examples:

```
>>> from sympy import *
>>> ask(Q.extended_real(oo))
True
>>> ask(Q.extended_real(2))
True
>>> ask(Q.extended_real(x), Q.real(x))
True
```

imaginary

Test that an expression belongs to the set of imaginary numbers, that is, it can be written as x*I, where x is real and I is the imaginary unit.

Examples:

```
>>> from sympy import *
>>> ask(Q.imaginary(2*I))
True
>>> x = Symbol('x')
>>> ask(Q.imaginary(x*I), Q.real(x))
True
```

infinitesimal

Test that an expression is equivalent to an infinitesimal number.

Examples:

```
>>> from sympy import *
>>> ask(Q.infinitesimal(1/oo))
True
>>> x, y = symbols('x,y')
>>> ask(Q.infinitesimal(2*x), Q.infinitesimal(x))
True
>>> ask(Q.infinitesimal(x*y), Q.infinitesimal(x) & Q.bounded(y))
True
```

integer

Test that an expression belongs to the set of integer numbers.

Examples:

```
>>> from sympy import *
>>> ask(Q.integer(2))
True
>>> ask(Q.integer(sqrt(2)))
False
>>> x = Symbol('x')
>>> ask(Q.integer(x/2), Q.even(x))
True
```

irrational

Test that an expression represents an irrational number.

Examples:

```
>>> from sympy import *
>>> ask(Q.irrational(pi))
True
>>> ask(Q.irrational(sqrt(2)))
True
>>> ask(Q.irrational(x*sqrt(2)), Q.rational(x))
True
```

rational

Test that an expression represents a rational number.

Examples:

```
>>> from sympy import *
>>> ask(Q.rational(Rational(3, 4)))
True
>>> x, y = symbols('x,y')
>>> ask(Q.rational(x/2), Q.integer(x))
True
>>> ask(Q.rational(x/y), Q.integer(x) & Q.integer(y))
True
```

negative

Test that an expression is less (strict) than zero.

Examples:

```
>>> from sympy import *
>>> ask(Q.negative(0.3))
False
>>> x = Symbol('x')
>>> ask(Q.negative(-x), Q.positive(x))
True
```

Remarks

negative numbers are defined as real numbers that are not zero nor positive, so complex numbers (with nontrivial imaginary coefficients) will return False for this predicate. The same applies to Q.positive.

positive

Test that a given expression is greater (strict) than zero.

Examples:

```
>>> from sympy import *
>>> ask(Q.positive(0.3))
True
>>> x = Symbol('x')
>>> ask(Q.positive(-x), Q.negative(x))
True
```

Remarks

see Remarks for negative

prime

Test that an expression represents a prime number.

Examples:

```
>>> from sympy import *
>>> ask(Q.prime(13))
True
```

Remarks: Use sympy.ntheory.isprime to test numeric values efficiently.

real

Test that an expression belongs to the field of real numbers.

Examples:

```
>>> from sympy import *
>>> ask(Q.real(sqrt(2)))
True
>>> x, y = symbols('x,y')
>>> ask(Q.real(x*y), Q.real(x) & Q.real(y))
True
```

odd

Test that an expression represents an odd number.

Examples:

```
>>> from sympy import *
>>> ask(Q.odd(3))
True
>>> n = Symbol('n')
>>> ask(Q.odd(2*n + 1), Q.integer(n))
True
```

nonzero

Test that an expression is not zero.

Examples:

```
>>> from sympy import *
>>> x = Symbol('x')
>>> ask(Q.nonzero(x), Q.positive(x) | Q.negative(x))
True
```

5.20.4 Design

Each time ask is called, the appropriate Handler for the current key is called. This is always a subclass of sympy.assumptions.AskHandler. It's classmethods have the name's 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.

5.20.5 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):
...         import math
...         return ask(Q.integer(math.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 types [2] 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
```

5.20.6 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.

5.20.7 Misc

You can find more examples in the in the form of test under directory sympy/assumptions/tests/

5.21 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

[2] New type must inherit from Basic, otherwise an exception will be raised. This is a bug and should be fixed.

can be included here. Currently SymPy has several functions and basic built-in methods for performing various types of rewriting.

5.21.1 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:

```
>>> 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 give 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
```

5.21.2 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)
⎛    ⎡  ________⎤⎞
⎝[], ⎣\╱ sin(x) ⎦⎠

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
⎛                ⎡  ________   ________⎤⎞
⎝[(x₀, sin(x))], ⎣\╱ x₀ + 4 ⋅\╱ x₀ + 5 ⎦⎠

>>> 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:

5.22 Series Expansions

The series module implements series expansions as a function and many related functions.

5.22.1 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 z0.

z0 can be any expression, including oo and -oo.

For dir="+" (default) it calculates the limit from the right (z->z0+) and for dir="-" the limit from the left (z->z0-). For infinite z0 (oo or -oo), the dir argument doesn't matter.

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, dir="+")
oo
>>> limit(1/x, x, 0, dir="-")
-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 $\lim_{x\to\infty} f(x) = \infty$ and similarly $\lim_{x\to\infty} g(x) = \infty$. 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 $\lim_{x\to\infty} \frac{f(x)^a}{g(x)^b} = \infty$. 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 $x \to \infty$. 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 ω, $g_1, g_2, \ldots$ are functions of x, $\lim_{x\to\infty} \omega = 0$ and $\omega \succ g_i$ for all i. Let $c_1, c_2, \cdots \in \mathbb{R}$ with $c_1 < c_2 < \ldots$.
>
> Then $\lim_{x\to\infty} \sum_i g_i \omega^{c_i} = \lim_{x\to\infty} g_1 \omega^{c_1}$.

For $g_1 = g$ and ω as above we also have the following easy result:

- $\lim_{x\to\infty} g\omega^c = 0$ for $c > 0$
- $\lim_{x\to\infty} g\omega^c = \pm\infty$ for $c < 0$, where the sign is determined by the (eventual) sign of g
- $\lim_{x\to\infty} g\omega^0 = \lim_{x\to\infty} g$.

Using these results yields the following strategy for computing $\lim_{x\to\infty} f(x)$:

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 $\lim_{x\to\infty} \omega = 0$.
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 $\lim_{x\to\infty} f(x)$, 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 z0 using the Gruntz algorithm.
>
> z0 can be any expression, including oo and -oo.
>
> For dir="+" (default) it calculates the limit from the right (z->z0+) and for dir="-" the limit from the left (z->z0-). For infinite z0 (oo or -oo), 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 Omega ... 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 Omega (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 e1, e2, e3, e4 or e1, e2, e4, e3. 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`**(**args*, ***kw_args*)
Returns (c0, e0) for e.

`sympy.series.gruntz.`**`calculate_series`**(*e*, *x*, *skip_abs=False*, *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`**(**args*, ***kw_args*)
Limit e(x) for x-> oo

`sympy.series.gruntz.`**`sign`**(**args*, ***kw_args*)
Returns a sign of an expression e(x) for x->oo.

```
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 arbitarily 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 SubsSet 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 (union, expsboth) [if f, g are of the same class].

class sympy.series.gruntz.**SubsSet**

Stores (expr, 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 expr->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 choosen to be exp(-x), i.e. d2. The correct rewriting then is:

```
exp(-w)/w + 1/w + x.
```

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

5.22.2 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 intuative 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 $x = x0$.

See the doctring of Expr.series() for complete details of this wrapper.

5.22.3 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 0 is currently supported. This is expressed in big O notation [R230] (page 1627).

The formal definition for the order of a function $g(x)$ about a point a is such that $g(x) = O(f(x))$ as $x \to a$ 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 $\lim_{x\to a} |g(x)/f(x)| < \infty$.

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 $O(x^5) = x^5/5! - x^7/7! + \cdots$. By the definition of O, for any $\delta > 0$ there is an M such that:

$$|x^5/5! - x^7/7! + ...| <= M|x^5| \text{ for } |x| < \delta$$

or by the alternate definition:

$$\lim_{x\to 0}|(x^5/5! - x^7/7! + ...)/x^5| < \infty$$

which surely is true, because

$$\lim_{x\to 0}|(x^5/5! - x^7/7! + ...)/x^5| = 1/5!$$

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 $x^3, x^4, \ldots$ 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(expr*f(x), x)` is `O(f(x), x)`

`O(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:

References

[R230] (page 1627)

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> O(x)
O(x)
>>> O(x)*x
O(x**2)
>>> O(x)-O(x)
O(x)
```

contains(**args*, ***kw_args*)

Return True if expr belongs to Order(self.expr, *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).

5.22.4 Series Acceleration

TODO

Reference

sympy.series.acceleration.**richardson**(*A, k, n, N*)

Calculate an approximation for lim k->oo A(k) using Richardson extrapolation with the terms A(n), A(n+1), ..., A(n+N+1). Choosing N ~= 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->oo A(k) using the n-term Shanks transformation S(A)(n). With m > 1, calculate the m-fold recursive Shanks transformation S(S(...S(A)...))(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 0.69314718055994530941723212 15.

5.22.5 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-x0) in the power series expansion about x=x0.

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].

5.23 Sets

5.23.1 Set

class sympy.core.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 for that.

Real intervals are represented by the Interval class and unions of sets by the Union class. The empty set is represented by the EmptySet class and available as a singleton as S.EmptySet.

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

complement

The complement of 'self'.

As a shortcut it is possible to use the '~' or '-' operators:

```
>>> from sympy import Interval

>>> Interval(0, 1).complement
(-oo, 0) U (1, oo)
>>> ~Interval(0, 1)
(-oo, 0) U (1, oo)
>>> -Interval(0, 1)
(-oo, 0) U (1, oo)
```

contains(*other*)

Returns True if 'other' is contained in 'self' as an element.

As a shortcut it is possible to use the 'in' operator:

```
>>> from sympy import Interval

>>> Interval(0, 1).contains(0.5)
True
>>> 0.5 in Interval(0, 1)
True
```

inf

The infimum of 'self'

```
>>> from sympy import Interval, Union

>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

intersect(*other*)

Returns the intersection of 'self' and 'other'.

```
>>> from sympy import Interval

>>> Interval(1, 3).intersect(Interval(1, 2))
[1, 2]
```

measure

The (Lebesgue) measure of 'self'

```
>>> from sympy import Interval, Union

>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

sort_key(*order=None*)
Give sort_key of infimum (if possible) else sort_key of the set.

subset(*other*)
Returns True if 'other' is a subset of 'self'.

```
>>> from sympy import Interval

>>> Interval(0, 1).contains(0)
True
>>> Interval(0, 1, left_open=True).contains(0)
False
```

sup
The supremum of 'self'

```
>>> from sympy import Interval, Union

>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

union(*other*)
Returns the union of 'self' and 'other'.

As a shortcut it is possible to use the '+' operator:

```
>>> from sympy import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
[0, 1] U [2, 3]
>>> Interval(0, 1) + Interval(2, 3)
[0, 1] U [2, 3]
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
(1, 2] U {3}
```

Similarly it is possible to use the '-' operator for set differences:

```
>>> Interval(0, 2) - Interval(0, 1)
(1, 2]
>>> Interval(1, 3) - FiniteSet(2)
[1, 2) U (2, 3]
```

Elementary Sets

5.23.2 Interval

class sympy.core.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

<http://en.wikipedia.org/wiki/Interval_(mathematics)>

Examples

```
>>> from sympy import Symbol, Interval

>>> Interval(0, 1)
[0, 1]
>>> Interval(0, 1, False, True)
[0, 1)

>>> a = Symbol('a', real=True)
>>> Interval(0, a)
[0, a]
```

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

as_relational(*symbol*)

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.

```
>>> 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.

```
>>> from sympy import Interval

>>> Interval(0, 1).start
0
```

left_open

True if 'self' is left-open.

```
>>> from sympy import Interval

>>> Interval(0, 1, left_open=True).left_open
True
>>> Interval(0, 1, left_open=False).left_open
False
```

right

The right end point of 'self'.

This property takes the same value as the 'sup' property.

```
>>> from sympy import Interval

>>> Interval(0, 1).end
1
```

right_open

True if 'self' is right-open.

```
>>> 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.

```
>>> from sympy import Interval

>>> Interval(0, 1).start
0
```

5.23.3 FiniteSet

class sympy.core.sets.**FiniteSet**

Represents a finite set of discrete numbers

References

http://en.wikipedia.org/wiki/Finite_set

Examples

```
>>> from sympy import FiniteSet
```

```
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
>>> 3 in FiniteSet(1, 2, 3, 4)
True
```

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

as_relational(*symbol*)
Rewrite a FiniteSet in terms of equalities and logic operators.

Compound Sets

5.23.4 Union

class sympy.core.sets.**Union**
Represents a union of sets as a Set.

See Also:

Intersection (page 1178)

References

<http://en.wikipedia.org/wiki/Union_(set_theory)>

Examples

```
>>> from sympy import Union, Interval

>>> Union(Interval(1, 2), Interval(3, 4))
[1, 2] U [3, 4]
```

The Union constructor will always try to merge overlapping intervals, if possible. For example:

```
>>> Union(Interval(1, 2), Interval(2, 3))
[1, 3]
```

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

as_relational(*symbol*)
Rewrite a Union in terms of equalities and logic operators.

static reduce(*args*)

Simplify a Union using known rules

We first start with global rules like 'Merge all FiniteSets'

Then we iterate through all pairs and ask the constituent sets if they can simplify themselves with any other constituent

5.23.5 Intersection

class sympy.core.sets.**Intersection**

Represents an intersection of sets as a Set.

See Also:

Union (page 1177)

References

<http://en.wikipedia.org/wiki/Intersection_(set_theory)>

Examples

```
>>> from sympy import Intersection, Interval
```

```
>>> Intersection(Interval(1, 3), Interval(2, 4))
[2, 3]
```

We often use the .intersect method

```
>>> Interval(1,3).intersect(Interval(2,4))
[2, 3]
```

Attributes

is_EmptySet	
is_UniversalSet	

as_relational(*symbol*)

Rewrite an Intersection in terms of equalities and logic operators

static reduce(*args*)

Simplify an intersection using known rules

We first start with global rules like 'if any empty sets return empty set' and 'distribute any unions'

Then we iterate through all pairs and ask the constituent sets if they can simplify themselves with any other constituent

5.23.6 ProductSet

class sympy.core.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

http://en.wikipedia.org/wiki/Cartesian_product

Examples

```
>>> from sympy import Interval, FiniteSet, ProductSet

>>> I = Interval(0, 5); S = FiniteSet(1, 2, 3)
>>> ProductSet(I, S)
[0, 5] x {1, 2, 3}

>>> (2, 2) in ProductSet(I, S)
True

>>> Interval(0, 1) * Interval(0, 1) # The unit square
[0, 1] x [0, 1]

>>> coin = FiniteSet('H', 'T')
>>> set(coin**2)
set([(H, H), (H, T), (T, H), (T, T)])
```

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

Singleton Sets

5.23.7 EmptySet

class sympy.core.sets.**EmptySet**

Represents the empty set. The empty set is available as a singleton as S.EmptySet.

See Also:

UniversalSet (page 1180)

References

http://en.wikipedia.org/wiki/Empty_set

Examples

```
>>> from sympy import S, Interval

>>> S.EmptySet
EmptySet()

>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet()
```

Attributes

is_Intersection	
is_UniversalSet	

5.23.8 UniversalSet

class sympy.core.sets.**UniversalSet**

Represents the set of all things. The universal set is available as a singleton as S.UniversalSet

See Also:

EmptySet (page 1179)

References

http://en.wikipedia.org/wiki/Universal_set

Examples

```
>>> from sympy import S, Interval

>>> S.UniversalSet
UniversalSet()

>>> Interval(1, 2).intersect(S.UniversalSet)
[1, 2]
```

Attributes

is_EmptySet	
is_Intersection	

Special Sets

5.23.9 Naturals

class sympy.sets.fancysets.**Naturals**

Represents the Natural Numbers. The Naturals are available as a singleton as S.Naturals

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> print iterable.next()
1
>>> print iterable.next()
2
>>> print iterable.next()
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

5.23.10 Integers

class sympy.sets.fancysets.**Integers**

Represents the Integers. The Integers are available as a singleton as S.Integers

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> print iterable.next()
0
>>> print iterable.next()
1
>>> print iterable.next()
```

```
-1
>>> print iterable.next()
2

>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

Attributes

is_EmptySet	
is_Intersection	
is_UniversalSet	

5.23.11 ImageSet

5.24 Simplify

5.24.1 simplify

sympy.simplify.simplify.**simplify**(*expr, ratio=1.7, measure=<function count_ops at 0x0000000004035128>, fu=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 1182) may actually result in some more complicated expression. The default `ratio=1.7` prevents more extreme cases: if (result

length)/(input length) > 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.

5.24.2 collect

`sympy.simplify.simplify.`**`collect`**(*expr, syms, func=None, evaluate=True, 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 1184), so user is expected to provide an expression is an appropriate form. This makes `collect()` (page 1184) 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 1196), `collect_sqrt` (page 1196), `rcollect` (page 1186)

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 1184) 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, x)

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
a*Derivative(f(x), x, x) + b*Derivative(f(x), x, x)

>>> 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, x)**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.simplify.`**`rcollect`**(*expr*, **vars*)

Recursively collect sums in an expression.

See Also:

`collect` (page 1184), `collect_const` (page 1196), `collect_sqrt` (page 1196)

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)
```

5.24.3 separate

sympy.simplify.simplify.**separate**(*expr*, *deep=False*, *force=False*)

Deprecated wrapper for `expand_power_base()`. Use that function instead.

5.24.4 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
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) == None
True
```

5.24.5 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.

5.24.6 rad_rationalize

sympy.simplify.simplify.**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.simplify import rad_rationalize
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

5.24.7 radsimp

sympy.simplify.simplify.**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 != +/-b*sqrt(c). (If there are no symbols, this assumptions 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.simplify import collect_sqrt
>>> from sympy.abc import a, b, c

>>> radsimp(1/(I + 1))
(1 - I)/2
>>> 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)

  ___       ___       ___       ___
  \/ 5 *a + \/ 5 *b - \/ 2 *x - \/ 2 *y
-----------------------------------------
   2                2      2               2
5*a  + 10*a*b + 5*b  - 2*x  - 4*x*y - 2*y

>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))

      ___       ___
      \/ 5 *(a + b) - \/ 2 *(x + y)
-----------------------------------------
   2                2      2               2
5*a  + 10*a*b + 5*b  - 2*x  - 4*x*y - 2*y
```

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))
```

5.24.8 ratsimp

sympy.simplify.simplify.**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)
```

5.24.9 fraction

sympy.simplify.simplify.**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
>>> fraction(2*x**(-y))
(2, x**y)

>>> fraction(exp(-x))
(1, exp(x))

>>> fraction(exp(-x), exact=True)
(exp(-x), 1)
```

5.24.10 trigsimp

sympy.simplify.simplify.**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
```

5.24.11 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
```

5.24.12 powsimp

sympy.simplify.simplify.**powsimp**(*expr, deep=False, combine='all', force=False, measure=<function count_ops at 0x0000000004035128>*)

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) -> 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))

5.24.13 combsimp

`sympy.simplify.simplify.`**`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 result is be given in terms of binomials and factorials.

The algorithm works by rewriting all combinatorial functions as expressions involving rising factorials (Pochhammer symbols) and applies recurrence relations and other transformations applicable to rising factorials, to reduce their arguments, possibly letting the resulting rising factorial to cancel. Rising factorials with the second argument being an integer are expanded into polynomial forms and finally all other rising factorial are rewritten in terms more familiar functions. If the initial expression contained any combinatorial functions, the result is expressed using binomial coefficients and gamma functions. If the initial expression consisted of gamma functions alone, the result is expressed in terms of gamma functions.

If the result is expressed using gamma functions, the following three additional steps are performed:

1. Reduce the number of gammas by applying the reflection theorem gamma(x)*gamma(1-x) == pi/sin(pi*x).
2. Reduce the number of gammas by applying the multiplication theorem gamma(x)*gamma(x+1/n)*...*gamma(x+(n-1)/n) == C*gamma(n*x).
3. Reduce the number of prefactors by absorbing them into gammas, where possible.

All transformation rules can be found (or was derived from) here:

1. http://functions.wolfram.com/GammaBetaErf/Pochhammer/17/01/02/
2. http://functions.wolfram.com/GammaBetaErf/Pochhammer/27/01/0005/

Examples

```
>>> from sympy.simplify import combsimp
>>> from sympy import factorial, binomial
>>> from sympy.abc import n, k

>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
(n + 1)/(k + 1)
```

5.24.14 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

5.24.15 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().

5.24.16 nsimplify

`sympy.simplify.simplify.`**`nsimplify`**(*expr*, *constants=[]*, *tolerance=None*, *full=False*, *rational=None*)

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).

See Also:

`sympy.core.function.nfloat` (page 166)

Examples

```
>>> from sympy import nsimplify, sqrt, GoldenRatio, exp, I, exp, 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
```

5.24.17 collect_sqrt

sympy.simplify.simplify.**collect_sqrt**(*expr*, *evaluate=True*)

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 = sqrt(-1), it is collected, too.

See Also:

`collect` (page 1184), `collect_const` (page 1196), `rcollect` (page 1186)

Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.simplify 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)
```

5.24.18 collect_const

sympy.simplify.simplify.**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.

See Also:

`collect` (page 1184), `collect_sqrt` (page 1196), `rcollect` (page 1186)

Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import a, s, x, y, z
>>> from sympy.simplify.simplify 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)
```

5.24.19 posify

sympy.simplify.simplify.**posify**(*eq*)

Return eq (with generic symbols made positive) and a restore dictionary.

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
>>> from sympy.abc import x
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(_x + n + p, {_x: x})
```

>> log(1/x).expand() # should be log(1/x) but it comes back as -log(x) log(1/x)

```
>>> log(posify(1/x)[0]).expand() # take [0] and ignore replacements
-log(_x)
>>> eq, rep = posify(1/x)
>>> log(eq).expand().subs(rep)
-log(x)
>>> posify([x, 1 + x])
([_x, _x + 1], {_x: x})
```

5.24.20 powdenest

sympy.simplify.simplify.**powdenest**(*eq, force=False, polar=False*)

Collect exponents on powers as assumptions allow.

Given (bbbe)**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)
>>> p = Symbol('p', positive=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(((p**(2*a))**(3*y))**x)
p**(6*a*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
```

5.24.21 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 1197)** replace all symbols with symbols having positive assumptions

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)
```

5.24.22 Square Root Denest

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://www.almaden.ibm.com/cs/people/fagin/symb85.pdf

[2] D. J. Jeffrey and A. D. Rich, 'Symplifying 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)
```

5.24.23 Common Subexpresion Elimination

cse

`sympy.simplify.cse_main.`**`cse`**(*exprs, symbols=None, optimizations=None, postprocess=None*)

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, optional

The (preprocessor, postprocessor) pairs. If not provided, `sympy.simplify.cse.cse_optimizations` is used.

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

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.

opt_cse

tree_cse

5.24.24 Hypergeometric Function Expansion

hyperexpand

`sympy.simplify.hyperexpand.`**`hyperexpand`**(*f, allow_hyper=False, rewrite='default'*)

Expand hypergeometric functions. If allow_hyper is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

Examples

```
>>> from sympy.simplify.hyperexpand import hyperexpand
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyperexpand(hyper([], [], z))
exp(z)
```

Non-hyperegeometric 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
```

5.24.25 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
```

5.24.26 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 1203) 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)
```

5.25 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 symplify module.

5.25.1 Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roach1996] (page 1627) and [Roach1997] (page 1627).

Recall that the hypergeometric function is (initially) defined as

$$ {}_pF_q\left(\left.\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix}\right| z\right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n}{(b_1)_n \dots (b_q)_n} \frac{z^n}{n!}. $$

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 conjuction 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 $A(a_i) = \frac{z}{a_i}\frac{\mathrm{d}}{dz} + 1$. 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 $B(b_j) = \frac{z}{b_j-1}\frac{\mathrm{d}}{dz} + 1$ and find $B(b_j)_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = {}_pF_q\left(\begin{matrix}a_p\\b_q-e_i\end{matrix}\middle|z\right)$. 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 $\frac{\mathrm{d}}{dz}{}_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = \frac{a_1\ldots a_p}{b_1\ldots b_q}{}_pF_q\left(\begin{matrix}a_p+1\\b_q+1\end{matrix}\middle|z\right)$, where a_p+1 is the vector $a_1+1, a_2+1, \ldots$ and similarly for b_q+1. Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: $\left[\frac{\mathrm{d}}{dz}\prod_{j=1}^q B(b_j) - \frac{a_1\ldots a_p}{(b_1-1)\ldots(b_q-1)}\prod_{i=1}^p A(a_i)\right]{}_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = 0$. 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: $\left[z\frac{\mathrm{d}}{dz}\prod_{j=1}^q\left(z\frac{\mathrm{d}}{dz}+b_j-1\right) - z\prod_{i=1}^p\left(z\frac{\mathrm{d}}{\mathrm{d}z}+a_i\right)\right]{}_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = 0$. 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 $z\frac{\mathrm{d}}{\mathrm{d}z}$ 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: $[XA(a_I) - r]_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = 0$, for some operator X and some constant r to be determined. If $r \neq 0$, then we can write this as $\frac{-1}{r}X_pF_q\left(\begin{matrix}a_p+e_I\\b_q\end{matrix}\middle|z\right) = {}_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right)$, 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 $z\frac{\mathrm{d}}{\mathrm{d}z} = a_IA(a_I) - a_I$. Observe also that all the operators $A(a_i)$, $B(b_j)$ and $z\frac{\mathrm{d}}{\mathrm{d}z}$ commute. We have $\prod_{i=1}^p\left(z\frac{\mathrm{d}}{\mathrm{d}z}+a_i\right) = \left(\prod_{i=1,i\neq I}^p\left(z\frac{\mathrm{d}}{\mathrm{d}z}+a_i\right)\right)a_IA(a_I)$, so this gives us the first half of X. The other half does not have such a nice expression. We find $z\frac{\mathrm{d}}{dz}\prod_{j=1}^q\left(z\frac{\mathrm{d}}{dz}+b_j-1\right) = (a_IA(a_I)-a_I)\prod_{j=1}^q(a_IA(a_I)-a_I+b_j-1)$. 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 $B(a_J)$, 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 ${}_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = {}_{p-1}F_{q-1}\left(\begin{matrix}a_p^*\\b_q^*\end{matrix}\middle|z\right)$, 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 $(z\frac{\mathrm{d}}{\mathrm{d}z})^k z^n = n^k z^n$. Combining these two remarks we find:

> If $a_I - b_J \in \mathbb{Z}_{>0}$, then there exists a polynomial $p(n) = p_0 + p_1 n + \ldots$ (of degree $a_I - b_J$) such that $\frac{(a_I)_n}{(b_J)_n} = p(n)$ and ${}_pF_q\left(\begin{matrix}a_p\\b_q\end{matrix}\middle|z\right) = \left(p_0 + p_1 z\frac{\mathrm{d}}{\mathrm{d}z} + p_2\left(z\frac{\mathrm{d}}{\mathrm{d}z}\right)^2 + \ldots\right){}_{p-1}F_{q-1}\left(\begin{matrix}a_p^*\\b_q^*\end{matrix}\middle|z\right)$.

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 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:

- ${}_rF_s\left(\begin{matrix} a_r^0 \\ b_s^0 \end{matrix}\middle| z\right) = CB$
- $z\frac{\mathrm{d}}{\mathrm{d}z}B = MB$.

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.

5.25.2 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

$$G(z) = \frac{1}{2\pi i}\int_L \frac{\prod_{j=1}^m \Gamma(b_j - s)\prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s)\prod_{j=n+1}^p \Gamma(a_j - s)} z^s \mathrm{d}s,$$

where L is a contour starting and ending at $+\infty$, enclosing all of the poles of $\Gamma(b_j - s)$ for $j = 1, \ldots, n$ 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 $b, b + k_1, \ldots, b + k_u$, with k_i non-negative integers. Enumerate similarly $\{a_j | a_j \equiv c \pmod 1, j > n\}$ as $b + l_1, \ldots, b + l_v$. Then $l_i > k_j$ for all i, j. For finite confluence, we need to assume $v \geq u$ for all such c.

Let $c_1, \ldots, c_w$ 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 $b, b + k_1, \ldots, b + k_u$. We will look at the a_j corresponding to $a + l_1, \ldots, a + l_u$. The other a_i are not treated specially. The corresponding gamma functions have poles at (potentially) $s = b + r$ for $r = 0, 1, \ldots$. For $r \geq l_u$, pole of the integrand is simple. We thus set

$$R(z) = \sum_{r=0}^{l_u - 1} res_{s=r+b}.$$

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.)

5.25.3 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 dummys `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 dummys `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:

$$C(z) = z \cdot {}_1F_2\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix}\middle| -\frac{\pi^2 z^4}{16}\right).$$

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 ${}_1F_2(\ldots|f(z))$ 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 ${}_1F_2(\ldots|z)$. 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)

$$\begin{aligned} f(g(w)) &= -\frac{\pi^2 g(w)^4}{16} \\ &= -\frac{\pi^2 g\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)^4}{16} \\ &= -\frac{\pi^2 \frac{2^4}{\sqrt{\pi}^4} \exp\left(\frac{i\pi}{4}\right)^4 w^{\frac{1}{4}}{}^4}{16} \\ &= -\exp\left(i\pi\right) w \\ &= w \end{aligned}$$

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

$$C(g(w)) = g(w) \cdot {}_1F_2\left(\left.\begin{matrix}\frac{1}{4} \\ \frac{1}{2}, \frac{5}{4}\end{matrix}\right| w\right).$$

and trivially

$${}_1F_2\left(\left.\begin{matrix}\frac{1}{4} \\ \frac{1}{2}, \frac{5}{4}\end{matrix}\right| w\right) = \frac{C(g(w))}{g(w)} = \frac{C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)}{\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third paramenter, `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 `addb(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 $\{B_0, B_1, \ldots\}$ 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 usefull pieces. In our case we find that $n = \max(1, 2+1) = 3$. For computing the derivatives, we have to use the operator $z\frac{\mathrm{d}}{\mathrm{d}z}$. The first basis element B_0 is set to the expression for ${}_1F_2$ from above:

$$B_0 = \frac{\sqrt{\pi}\exp\left(-\frac{\imath\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}}\exp\left(\frac{\imath\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}}$$

Next we compute $z\frac{\mathrm{d}}{\mathrm{d}z}B_0$. 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

$$B_1' = -\frac{1}{4}\frac{\sqrt{\pi}\exp\left(-\frac{\imath\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}}\exp\left(\frac{\imath\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} + \frac{1}{4}\cosh\left(2\sqrt{z}\right)$$

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)
>>> 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/
```

which can be printed as

$$B_2' = \frac{1}{16}\frac{\sqrt{\pi}\exp\left(-\frac{\imath\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}}\exp\left(\frac{\imath\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16}\cosh\left(2\sqrt{z}\right) + \frac{1}{4}\sinh\left(2\sqrt{z}\right)\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose B_1 and B_2 as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{\pi}\exp\left(-\frac{\imath\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}}\exp\left(\frac{\imath\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} \\ \cosh\left(2\sqrt{z}\right) \\ \sinh\left(2\sqrt{z}\right)\sqrt{z} \end{pmatrix}$$

(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 ${}_pF_q\left(\cdots|z\right) = CB$ 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 $z\frac{\mathrm{d}}{\mathrm{d}z}B = MB$ holds. This is easy. We already computed the first part $z\frac{\mathrm{d}}{\mathrm{d}z}B_0$ 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`.

5.25.4 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 1627) and [Prudnikov1990] (page 1627). They are all tested numerically.

$${}_0F_0\left(\middle|z\right) = e^z$$

$${}_1F_0\left(a\middle|z\right) = \left(-z + 1\right)^{-a}$$

$$ {}_2F_1\left(\left.\begin{matrix} a, a - \frac{1}{2} \\ 2a \end{matrix}\right| z\right) = 2^{2a-1} \left(\sqrt{-z+1} + 1\right)^{-2a+1} $$

$$ {}_2F_1\left(\left.\begin{matrix} 1, 1 \\ 2 \end{matrix}\right| z\right) = -\frac{1}{z} \log\left(-z+1\right) $$

$$ {}_2F_1\left(\left.\begin{matrix} \frac{1}{2}, 1 \\ \frac{3}{2} \end{matrix}\right| z\right) = \frac{1}{\sqrt{z}} \operatorname{atanh}\left(\sqrt{z}\right) $$

$$ {}_2F_1\left(\left.\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ \frac{3}{2} \end{matrix}\right| z\right) = \frac{1}{\sqrt{z}} \operatorname{asin}\left(\sqrt{z}\right) $$

$$ {}_2F_1\left(\left.\begin{matrix} a, a + \frac{1}{2} \\ \frac{1}{2} \end{matrix}\right| z\right) = \frac{1}{2} \left(\sqrt{z}+1\right)^{-2a} + \frac{1}{2} \left(-\sqrt{z}+1\right)^{-2a} $$

$$ {}_2F_1\left(\left.\begin{matrix} a, -a \\ \frac{1}{2} \end{matrix}\right| z\right) = \cos\left(2a \operatorname{asin}\left(\sqrt{z}\right)\right) $$

$$ {}_2F_1\left(\left.\begin{matrix} 1, 1 \\ \frac{3}{2} \end{matrix}\right| z\right) = \frac{\operatorname{asin}\left(\sqrt{z}\right)}{\sqrt{z}\sqrt{-z+1}} $$

$$ {}_2F_1\left(\left.\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ 1 \end{matrix}\right| z\right) = \frac{2K\left(z\right)}{\pi} $$

$$ {}_2F_1\left(\left.\begin{matrix} -\frac{1}{2}, \frac{1}{2} \\ 1 \end{matrix}\right| z\right) = \frac{2E\left(z\right)}{\pi} $$

$$ {}_3F_2\left(\left.\begin{matrix} -\frac{1}{2}, 1, 1 \\ \frac{1}{2}, 2 \end{matrix}\right| z\right) = -\frac{2\sqrt{z}}{3} \operatorname{atanh}\left(\sqrt{z}\right) + \frac{2}{3} - \frac{1}{3z} \log\left(-z+1\right) $$

$$ {}_3F_2\left(\left.\begin{matrix} -\frac{1}{2}, 1, 1 \\ 2, 2 \end{matrix}\right| z\right) = \left(\frac{4}{9} - \frac{16}{9z}\right) \sqrt{-z+1} + \frac{4}{3z} \log\left(\frac{1}{2}\sqrt{-z+1} + \frac{1}{2}\right) + \frac{16}{9z} $$

$$ {}_1F_1\left(\begin{matrix}1\\b\end{matrix}\middle| z\right) = z^{-b+1}\left(b-1\right)e^{z}\gamma\left(b-1,z\right) $$

$$ {}_1F_1\left(\begin{matrix}a\\2a\end{matrix}\middle| z\right) = 4^{a-\frac{1}{2}}z^{-a+\frac{1}{2}}e^{\frac{z}{2}}I_{a-\frac{1}{2}}\left(\frac{z}{2}\right)\Gamma\left(a+\frac{1}{2}\right) $$

$$ {}_1F_1\left(\begin{matrix}a\\a+1\end{matrix}\middle| z\right) = a\left(ze^{i\pi}\right)^{-a}\gamma\left(a,ze^{i\pi}\right) $$

$$ {}_1F_1\left(\begin{matrix}-\frac{1}{2}\\\frac{1}{2}\end{matrix}\middle| z\right) = \sqrt{z}i\sqrt{\pi}\,\mathrm{erf}\left(\sqrt{z}i\right)+e^{z} $$

$$ {}_1F_2\left(\begin{matrix}1\\\frac{3}{4},\frac{5}{4}\end{matrix}\middle| z\right) = \frac{\sqrt{\pi}e^{-\frac{i\pi}{4}}}{2\sqrt[4]{z}}\left(i\sinh\left(2\sqrt{z}\right)S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}}e^{\frac{i\pi}{4}}\right)+\cosh\left(2\sqrt{z}\right)C\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}}e^{\frac{i\pi}{4}}\right)\right) $$

$$ {}_2F_2\left(\begin{matrix}\frac{1}{2},a\\\frac{3}{2},a+1\end{matrix}\middle| z\right) = -\frac{ai\sqrt{\pi}\sqrt{\frac{1}{z}}}{2a-1}\,\mathrm{erf}\left(\sqrt{z}i\right)-\frac{a\left(ze^{i\pi}\right)^{-a}}{2a-1}\gamma\left(a,ze^{i\pi}\right) $$

$$ {}_2F_2\left(\begin{matrix}1,1\\2,2\end{matrix}\middle| z\right) = \frac{1}{z}\left(-\log\left(z\right)+\mathrm{Ei}\left(z\right)\right)-\frac{\gamma}{z} $$

$$ {}_0F_1\left(\begin{matrix}\\\frac{1}{2}\end{matrix}\middle| z\right) = \cosh\left(2\sqrt{z}\right) $$

$$ {}_0F_1\left(\begin{matrix}\\b\end{matrix}\middle| z\right) = z^{-\frac{b}{2}+\frac{1}{2}}I_{b-1}\left(2\sqrt{z}\right)\Gamma\left(b\right) $$

$$ {}_0F_3\left(\begin{matrix}\\\frac{1}{2},a,a+\frac{1}{2}\end{matrix}\middle| z\right) = 2^{-2a}z^{-\frac{a}{2}+\frac{1}{4}}\left(I_{2a-1}\left(4\sqrt[4]{z}\right)+J_{2a-1}\left(4\sqrt[4]{z}\right)\right)\Gamma\left(2a\right) $$

$$ {}_0F_3\left(\begin{matrix}\\a,a+\frac{1}{2},2a\end{matrix}\middle| z\right) = \left(2\sqrt{z}e^{\frac{i\pi}{2}}\right)^{-2a+1}I_{2a-1}\left(2\sqrt{2}\sqrt[4]{z}e^{\frac{i\pi}{4}}\right)J_{2a-1}\left(2\sqrt{2}\sqrt[4]{z}e^{\frac{i\pi}{4}}\right)\Gamma^{2}\left(2a\right) $$

$$ {}_1F_2\left(\begin{matrix} a \\ a-\frac{1}{2}, 2a \end{matrix}\middle| z\right) = 2\cdot 4^{a-1} z^{-a+1} I_{a-\frac{3}{2}}\left(\sqrt{z}\right) I_{a-\frac{1}{2}}\left(\sqrt{z}\right) \Gamma\left(a-\frac{1}{2}\right) \Gamma\left(a+\frac{1}{2}\right) - 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} I_{a-\frac{1}{2}}^2\left(\sqrt{z}\right) \Gamma^2\left(a+\frac{1}{2}\right) $$

$$ {}_1F_2\left(\begin{matrix} \frac{1}{2} \\ b, -b+2 \end{matrix}\middle| z\right) = \frac{\pi I_{-b+1}\left(\sqrt{z}\right) I_{b-1}\left(\sqrt{z}\right)}{\sin\left(b\pi\right)} \left(-b+1\right) $$

$$ {}_1F_2\left(\begin{matrix} \frac{1}{2} \\ \frac{3}{2}, \frac{3}{2} \end{matrix}\middle| z\right) = \frac{1}{2\sqrt{z}} \operatorname{Shi}\left(2\sqrt{z}\right) $$

$$ {}_1F_2\left(\begin{matrix} \frac{3}{4} \\ \frac{3}{2}, \frac{7}{4} \end{matrix}\middle| z\right) = \frac{3\sqrt{\pi}}{4z^{\frac{3}{4}}} e^{-\frac{3\pi}{4}i} S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) $$

$$ {}_1F_2\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix}\middle| z\right) = \frac{\sqrt{\pi} e^{-\frac{i\pi}{4}}}{2\sqrt[4]{z}} C\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) $$

$$ {}_2F_3\left(\begin{matrix} a, a+\frac{1}{2} \\ 2a, b, 2a-b+1 \end{matrix}\middle| z\right) = \left(\frac{\sqrt{z}}{2}\right)^{-2a+1} I_{2a-b}\left(\sqrt{z}\right) I_{b-1}\left(\sqrt{z}\right) \Gamma\left(b\right) \Gamma\left(2a-b+1\right) $$

$$ {}_2F_3\left(\begin{matrix} 1, 1 \\ 2, 2, \frac{3}{2} \end{matrix}\middle| z\right) = \frac{1}{z}\left(-\log\left(2\sqrt{z}\right) + \operatorname{Chi}\left(2\sqrt{z}\right)\right) - \frac{\gamma}{z} $$

$$ {}_3F_3\left(\begin{matrix} 1, 1, a \\ 2, 2, a+1 \end{matrix}\middle| z\right) = \frac{a\left(-z\right)^{-a}}{\left(a-1\right)^2}\left(\Gamma\left(a\right) - \Gamma\left(a, -z\right)\right) + \frac{a}{z\left(a^2-2a+1\right)}\left(-a+1\right)\left(\log\left(-z\right) + \mathrm{E}_1\left(-z\right) + \gamma\right) - \frac{ae^z}{z\left(a^2-2a+1\right)} + $$

5.25.5 References

5.26 Statistics

Note: This module has been deprecated. See the `stats` (page 1222) module.

The `statistics` module in SymPy implements standard probability distributions and related tools. Its contents can be imported with the following statement:

```
>>> from sympy import *
>>> from sympy.statistics import *
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
```

5.26.1 Normal distributions

`Normal(mu, sigma)` creates a normal distribution with mean value `mu` and standard deviation `sigma`. The `Normal` class defines several useful methods and properties. Various properties can be accessed directly as follows:

```
>>> N = Normal(0, 1)
>>> N.mean
0
>>> N.median
0
>>> N.variance
1
>>> N.stddev
1
```

You can generate random numbers from the desired distribution with the `random` method:

```
>>> N = Normal(10, 5)
>>> N.random()
4.914375200829805834246144514
>>> N.random()
11.84331557474637897087177407
>>> N.random()
17.22474580071733640806996846
>>> N.random()
9.864643097429464546621602494
```

The probability density function (pdf) and cumulative distribution function (cdf) of a distribution can be computed, either in symbolic form or for particular values:

```
>>> N = Normal(1, 1)
>>> x = Symbol('x')
>>> N.pdf(1)
  ___
 \/ 2
--------
    ____
2*\/ pi
>>> N.pdf(3).evalf()
0.0539909665131880
>>> N.cdf(x)
   /  ___         \
   |\/ 2 *(x - 1)|
erf|-------------|
   \      2      /   1
----------------- + -
        2            2
>>> N.cdf(-oo), N.cdf(1), N.cdf(oo)
(0, 1/2, 1)
>>> N.cdf(5).evalf()
0.999968328758167
```

The method `probability` gives the total probability on a given interval (a convenient alternative syntax for cdf(b)-cdf(a)):

```
>>> N = Normal(0, 1)
>>> N.probability(-oo, 0)
1/2
```

```
>>> N.probability(-1, 1)
   /  ___\
   |\/ 2 |
erf|-----|
   \  2  /
>>> N.probability(-1, 1).evalf()
0.682689492137086
```

You can also generate a symmetric confidence interval from a given desired confidence level (given as a fraction 0-1). For the normal distribution, 68%, 95% and 99.7% confidence levels respectively correspond to approximately 1, 2 and 3 standard deviations:

```
>>> N = Normal(0, 1)
>>> N.confidence(0.68)
(-0.994457883209753, 0.994457883209753)
>>> N.confidence(0.95)
(-1.95996398454005, 1.95996398454005)
>>> N.confidence(0.997)
(-2.96773792534178, 2.96773792534178)
```

Plug the interval back in to see that the value is correct:

```
>>> N.probability(*N.confidence(0.95)).evalf()
0.950000000000000
```

5.26.2 Other distributions

Besides the normal distribution, uniform continuous distributions are also supported. `Uniform(a, b)` represents the distribution with uniform probability on the interval [a, b] and zero probability everywhere else. The `Uniform` class supports the same methods as the `Normal` class.

Additional distributions, including support for arbitrary user-defined distributions, are planned for the future.

5.26.3 API Reference

Sample

class `sympy.statistics.distributions.`**`Sample`**

Sample([x1, x2, x3, ...]) represents a collection of samples. Sample parameters like mean, variance and stddev can be accessed as properties. The sample will be sorted.

Examples

```
>>> from sympy.statistics.distributions import Sample
>>> Sample([0, 1, 2, 3])
Sample([0, 1, 2, 3])
>>> Sample([8, 3, 2, 4, 1, 6, 9, 2])
Sample([1, 2, 2, 3, 4, 6, 8, 9])
>>> s = Sample([1, 2, 3, 4, 5])
>>> s.mean
3
```

```
>>> s.stddev
sqrt(2)
>>> s.median
3
>>> s.variance
2
```

Continuous Probability Distributions

class sympy.statistics.distributions.**ContinuousProbability**

Base class for continuous probability distributions

probability(*s*, *a*, *b*)

Calculate the probability that a random number x generated from the distribution satisfies a <= x <= b

Examples

```
>>> from sympy.statistics import Normal
>>> from sympy.core import oo
>>> Normal(0, 1).probability(-1, 1)
erf(sqrt(2)/2)
>>> Normal(0, 1).probability(1, oo)
-erf(sqrt(2)/2)/2 + 1/2
```

random(*s*, *n=None*)

random() – generate a random number from the distribution. random(n) – generate a Sample of n random numbers.

Examples

```
>>> from sympy.statistics import Uniform
>>> x = Uniform(1, 5).random()
>>> x < 5 and x > 1
True
>>> x = Uniform(-4, 2).random()
>>> x < 2 and x > -4
True
```

class sympy.statistics.distributions.**Normal**(*mu*, *sigma*)

Normal(mu, sigma) represents the normal or Gaussian distribution with mean value mu and standard deviation sigma.

Examples

```
>>> from sympy.statistics import Normal
>>> from sympy import oo
>>> N = Normal(1, 2)
>>> N.mean
1
>>> N.variance
```

```
4
>>> N.probability(-oo, 1)   # probability on an interval
1/2
>>> N.probability(1, oo)
1/2
>>> N.probability(-oo, oo)
1
>>> N.probability(-1, 3)
erf(sqrt(2)/2)
>>> _.evalf()
0.682689492137086
```

cdf(*s*, *x*)

Return the cumulative density function as an expression in x

Examples

```
>>> from sympy.statistics import Normal
>>> Normal(1, 2).cdf(0)
-erf(sqrt(2)/4)/2 + 1/2
>>> from sympy.abc import x
>>> Normal(1, 2).cdf(x)
erf(sqrt(2)*(x - 1)/4)/2 + 1/2
```

confidence(*s*, *p*)

Return a symmetric (p*100)% confidence interval. For example, p=0.95 gives a 95% confidence interval. Currently this function only handles numerical values except in the trivial case p=1.

For example, one standard deviation:

```
>>> from sympy.statistics import Normal
>>> N = Normal(0, 1)
>>> N.confidence(0.68)
(-0.994457883209753, 0.994457883209753)
>>> N.probability(*_).evalf()
0.680000000000000
```

Two standard deviations:

```
>>> N = Normal(0, 1)
>>> N.confidence(0.95)
(-1.95996398454005, 1.95996398454005)
>>> N.probability(*_).evalf()
0.950000000000000
```

static fit(*sample*)

Create a normal distribution fit to the mean and standard deviation of the given distribution or sample.

Examples

```
>>> from sympy.statistics import Normal
>>> Normal.fit([1,2,3,4,5])
Normal(3, sqrt(2))
>>> from sympy.abc import x, y
```

```
>>> Normal.fit([x, y])
Normal(x/2 + y/2, sqrt((-x/2 + y/2)**2/2 + (x/2 - y/2)**2/2))
```

pdf(*s*, *x*)

Return the probability density function as an expression in x

Examples

```
>>> from sympy.statistics import Normal
>>> Normal(1, 2).pdf(0)
sqrt(2)*exp(-1/8)/(4*sqrt(pi))
>>> from sympy.abc import x
>>> Normal(1, 2).pdf(x)
sqrt(2)*exp(-(x - 1)**2/8)/(4*sqrt(pi))
```

class sympy.statistics.distributions.**Uniform**(*a*, *b*)

Uniform(a, b) represents a probability distribution with uniform probability density on the interval [a, b] and zero density everywhere else.

cdf(*s*, *x*)

Return the cumulative density function as an expression in x

Examples

```
>>> from sympy.statistics import Uniform
>>> Uniform(1, 5).cdf(2)
1/4
>>> Uniform(1, 5).cdf(4)
3/4
```

confidence(*s*, *p*)

Generate a symmetric (p*100)% confidence interval.

```
>>> from sympy import Rational
>>> from sympy.statistics import Uniform
>>> U = Uniform(1, 2)
>>> U.confidence(1)
(1, 2)
>>> U.confidence(Rational(1,2))
(5/4, 7/4)
```

static fit(*sample*)

Create a uniform distribution fit to the mean and standard deviation of the given distribution or sample.

Examples

```
>>> from sympy.statistics import Uniform
>>> Uniform.fit([1, 2, 3, 4, 5])
Uniform(-sqrt(6) + 3, sqrt(6) + 3)
>>> Uniform.fit([1, 2])
Uniform(-sqrt(3)/2 + 3/2, sqrt(3)/2 + 3/2)
```

pdf(*s*, *x*)

Return the probability density function as an expression in x

Examples

```
>>> from sympy.statistics import Uniform
>>> Uniform(1, 5).pdf(1)
1/4
>>> Uniform(2, 4).pdf(2)
1/2
```

class sympy.statistics.distributions.**PDF**(*func, (x, a, b)) represents continuous probability distribution with probability distribution function func(x) on interval (a, b)*

If func is not normalized so that integrate(func, (x, a, b)) == 1, it can be normalized using PDF.normalize() method

Examples

```
>>> from sympy import Symbol, exp, oo
>>> from sympy.statistics.distributions import PDF
>>> from sympy.abc import x
>>> a = Symbol('a', positive=True)

>>> exponential = PDF(exp(-x/a)/a, (x,0,oo))
>>> exponential.pdf(x)
exp(-x/a)/a
>>> exponential.cdf(x)
1 - exp(-x/a)
>>> exponential.mean
a
>>> exponential.variance
a**2
```

cdf(*x*)

Return the cumulative density function as an expression in x

Examples

```
>>> from sympy.statistics.distributions import PDF
>>> from sympy import exp, oo
>>> from sympy.abc import x, y
>>> PDF(exp(-x/y), (x,0,oo)).cdf(4)
y - y*exp(-4/y)
>>> PDF(2*x + y, (x, 10, oo)).cdf(0)
-10*y - 100
```

normalize()

Normalize the probability distribution function so that integrate(self.pdf(x), (x, a, b)) == 1

Examples

```
>>> from sympy import Symbol, exp, oo
>>> from sympy.statistics.distributions import PDF
>>> from sympy.abc import x
>>> a = Symbol('a', positive=True)

>>> exponential = PDF(exp(-x/a), (x,0,oo))
>>> exponential.normalize().pdf(x)
exp(-x/a)/a
```

transform(*func, var*)

Return a probability distribution of random variable func(x) currently only some simple injective functions are supported

Examples

```
>>> from sympy.statistics.distributions import PDF
>>> from sympy import oo
>>> from sympy.abc import x, y
>>> PDF(2*x + y, (x, 10, oo)).transform(x, y)
PDF(0, ((_w,), x, x))
```

5.27 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

5.27.1 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
```

5.27.2 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)
{a: 1/3, b: 1/3, c: 1/3}

>>> Y = DiscreteUniform('Y', range(5)) # distribution over a range
>>> density(Y)
{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)
{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)
{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)
{0: 1/4, 1: 3/4}
```

```
>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails') # A fair coin toss
>>> density(X)
{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 gettings "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)
{H: 1/2, T: 1/2}

>>> C2 = Coin('C2', Rational(3, 5)) # An unfair coin
>>> density(C2)
{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)
{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)
{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)
2.00000000000000
>>> P(X>=2)
0.700000000000000
```

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

[R231] (page 1627)

Examples

```
>>> from sympy.stats import Arcsin, density
>>> 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)))
```

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 distrubtion 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

[R232] (page 1627), [R233] (page 1627)

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)
/                  /  z  \\             /  z  \            2/  z  \
|        2*beta*log|-----||  - alpha*log|-----| - beta*log  |-----|
|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

$$f(x) := \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\mathrm{B}(\alpha,\beta)}$$

with $x \in [0,1]$.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

Returns **A RandomSymbol.** :

References

[R234] (page 1627), [R235] (page 1627)

Examples

```
>>> from sympy.stats import Beta, density, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> 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)        *gamma(alpha + beta)
-----------------------------------------------
            gamma(alpha)*gamma(beta)

>>> 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

[R236] (page 1627), [R237] (page 1627)

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)             *gamma(alpha + beta)
----------------------------------------------------
              gamma(alpha)*gamma(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

[R238] (page 1627), [R239] (page 1627)

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

[R240] (page 1627), [R241] (page 1627)

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 363).

Parameters **k** : A positive Integer, $k > 0$, the number of degrees of freedom

l : Shift parameter

Returns **A RandomSymbol.** :

References

[R242] (page 1627)

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

$$f(x) := \frac{1}{2^{\frac{k}{2}}\Gamma\left(\frac{k}{2}\right)}x^{\frac{k}{2}-1}e^{-\frac{x}{2}}$$

with $x \geq 0$.

Parameters **k** : A positive Integer, $k > 0$, the number of degrees of freedom

Returns **A RandomSymbol.** :

References

[R243] (page 1627), [R244] (page 1628)

Examples

```
>>> from sympy.stats import ChiSquared, density, E, variance
>>> from sympy import Symbol, simplify, combsimp, 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)

>>> combsimp(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

[R245] (page 1628)

Examples

```
>>> from sympy.stats import Dagum, density
>>> 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
```

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

[R246] (page 1628), [R247] (page 1628)

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)
/  k*lowergamma(k, 0)   k*lowergamma(k, l*z)
|- ------------------ + --------------------  for z >= 0
<     gamma(k + 1)          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

[R248] (page 1628), [R249] (page 1628)

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

$$f(x) := \frac{\sqrt{\frac{(d_1 x)^{d_1} d_2^{d_2}}{(d_1 x + d_2)^{d_1 + d_2}}}}{x \mathrm{B}\left(\frac{d_1}{2}, \frac{d_2}{2}\right)}$$

with $x > 0$.

Parameters **d1** : $d_1 > 0$ a parameter

d2 : $d_2 > 0$ a parameter

Returns **A RandomSymbol.** :

References

[R250] (page 1628), [R251] (page 1628)

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       /d1   d2\
d2  *\/  (d1*z)  *(d1*z + d2)         *gamma|-- + --|
                                            \2    2 /
-----------------------------------------------------
                        /d1\      /d2\
                  z*gamma|--|*gamma|--|
                        \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

$$f(x) := \frac{2d_1^{d_1/2}d_2^{d_2/2}}{\mathrm{B}(d_1/2, d_2/2)}\frac{e^{d_1 z}}{(d_1e^{2z} + d_2)^{(d_1+d_2)/2}}$$

Parameters **d1** : $d_1 > 0$, degree of freedom

d2 : $d_2 > 0$, degree of freedom

Returns **A RandomSymbol.** :

References

[R252] (page 1628), [R253] (page 1628)

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
    d1   d2                 - -- - --
    --   --                   2    2
    2    2  /    2*z     \           d1*z      /d1   d2\
2*d1  *d2  *\d1*e    + d2/         *e    *gamma|-- + --|
                                                \2    2 /
-------------------------------------------------------
                        /d1\      /d2\
                  gamma|--|*gamma|--|
                        \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

$$f(x) := \frac{\alpha}{s}\left(\frac{x-m}{s}\right)^{-1-\alpha} e^{-(\frac{x-m}{s})^{-\alpha}}$$

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

[R254] (page 1628)

Examples

```
>>> from sympy.stats import Frechet, density, E, std
>>> 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
```

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

[R255] (page 1628), [R256] (page 1628)

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, -----|
|  k*lowergamma(k, 0)               \   theta/
<- ------------------ + ----------------------  for z >= 0
|     gamma(k + 1)           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

[R257] (page 1628)

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)
```

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

[R258] (page 1628)

Examples

```
>>> from sympy.stats import Kumaraswamy, density, E, variance
>>> 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/
```

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

[R259] (page 1628), [R260] (page 1628)

Examples

```
>>> from sympy.stats import Laplace, density
>>> 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)
```

sympy.stats.**Logistic**(*name*, *mu*, *s*)

Create a continuous random variable with a logistic distribution.

The density of the logistic distribution is given by

$$f(x) := \frac{e^{-(x-\mu)/s}}{s\left(1+e^{-(x-\mu)/s}\right)^2}$$

Parameters **mu** : Real number, the location (mean)

s : Real number, $s > 0$ a scale

Returns **A RandomSymbol.** :

References

[R261] (page 1628), [R262] (page 1628)

Examples

```
>>> from sympy.stats import Logistic, density
>>> 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)
```

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

[R263] (page 1628), [R264] (page 1628)

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

[R265] (page 1628), [R266] (page 1628)

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

[R267] (page 1628)

Examples

```
>>> from sympy.stats import Nakagami, density, E, variance
>>> 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
         omega*gamma (mu + 1/2)
omega - -----------------------
        gamma(mu)*gamma(mu + 1)
```

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

[R268] (page 1628), [R269] (page 1628)

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

[R270] (page 1628), [R271] (page 1628)

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

[R272] (page 1628)

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)
/                 2
|   /  a   b    \
```

```
|12*|- - - - + z|
|   \  2   2    /
<----------------  for And(a <= z, z <= b)
|            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

[R273] (page 1628)

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, mu - s <= z)
|         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

[R274] (page 1628), [R275] (page 1628)

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

[R276] (page 1628), [R277] (page 1628)

Examples

```
>>> from sympy.stats import StudentT, density, E, variance
>>> 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 |              /nu   1\
|1 + --|        *gamma|-- + -|
\    nu/              \2    2/
-----------------------------
        ____  ____     /nu\
   \/ pi *\/ nu *gamma|--|
                      \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

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \le x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \le b, \\ 0 & \text{for } b < x. \end{cases}$$

Parameters **a** : Real number, $a \in (-\infty, \infty)$

b : Real number, $a < b$

c : Real number, $a \le c \le b$

Returns **A RandomSymbol.** :

References

[R278] (page 1628), [R279] (page 1628)

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, z < c)
|(-a + b)*(-a + c)
|
|       2
|     ------              for z = c
<     -a + b
|
|   2*b - 2*z
|---------------   for And(z <= b, 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

[R280] (page 1628), [R281] (page 1628)

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), And(a <= z, z <= b)), (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

>>> simplify(skewness(X))
0
```

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

[R282] (page 1629), [R283] (page 1629)

Examples

```
>>> from sympy.stats import UniformSum, density
>>> 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)
  ___,
  \   ‘
   \         k         n - 1 /n\
    )    (-1) *(-k + z)     *| |
   /                         \k/
  /__,
 k = 0
--------------------------------
            (n - 1)!
```

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

[R284] (page 1629), [R285] (page 1629)

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

[R286] (page 1629), [R287] (page 1629)

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

[R288] (page 1629), [R289] (page 1629)

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=(-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
```

5.27.3 Interface

sympy.stats.**P**(*condition, given_condition=None, numsamples=None, **kwargs*)

Probability that a condition is true, optionally given a second condition

Parameters **expr** : Relational containing RandomSymbols

The condition of which you want to compute the probability

given_condition : Relational 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

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 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
```

`sympy.stats.E`(*expr, condition=None, numsamples=None, **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
```

`sympy.stats.density`(*expr, condition=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.

Examples

```
>>> from sympy.stats import density, Die, Normal
>>> from sympy import Symbol
```

```
>>> D = Die('D', 6)
>>> X = Normal('x', 0, 1)

>>> density(D)
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
>>> density(2*D)
{2: 1/6, 4: 1/6, 6: 1/6, 8: 1/6, 10: 1/6, 12: 1/6}
>>> density(X)
Lambda(x, sqrt(2)*exp(-x**2/2)/(2*sqrt(pi)))
```

sympy.stats.**given**(*expr, condition=None, **kwargs*)

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)
{4: 1/3, 5: 1/3, 6: 1/3}
```

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: And(-1 < x, x < 1)

>>> where(X**2<1).set
(-1, 1)

>>> where(And(D1<=D2 , D2<3))
Domain: Or(And(a == 1, b == 1), And(a == 1, b == 2), And(a == 2, 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)
```

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]
```

5.27.4 Mechanics

SymPy Stats employs a relatively complex class hierarchy.

`RandomDomain`s 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

5.28 ODE

5.28.1 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions (unlike Hint Functions (page 1263), below) are intended for use by ordinary

users of SymPy.

dsolve()

sympy.solvers.ode.**dsolve**(*eq*, *func=None*, *hint='default'*, *simplify=True*, ***kwargs*)

Solves any (supported) kind of ordinary differential equation.

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 1284) docstring for supported methods). This can either be an `Equality` (page 135), 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 1259). See Hints below for more options that you can use for hint.

`simplify` enables simplification by `odesimp()` (page 1264). 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.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to `dsolve()` (page 1257):

`default`: This uses whatever hint is returned first by `classify_ode()` (page 1259). This is the default argument to `dsolve()` (page 1257).

`all`: To make `dsolve()` (page 1257) 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 1305) 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 1259) 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 1259).

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 1257) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than `all`, because `integrate()` (page 109) is an expensive routine.

best: To have `dsolve()` (page 1257) 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. unevaluatable integrals), and which one is the shortest in size.

See also the `classify_ode()` (page 1259) docstring for more info on hints, and the `ode` (page 1284) 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 1257).
- `dsolve()` (page 1257) always returns an `Equality` (page 135) 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`.

Examples

```
>>> from sympy import Function, dsolve, Eq, Derivative, sin, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
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='separable_reduced')
f(x) == C1/(C2*x - 1)
>>> dsolve(eq, hint='1st_exact')
[f(x) == -acos(C1/cos(x)) + 2*pi, f(x) == acos(C1/cos(x))]
>>> dsolve(eq, hint='almost_linear')
[f(x) == -acos(-sqrt(C1/cos(x)**2)) + 2*pi, f(x) == -acos(sqrt(C1/cos(x)**2)) + 2*pi,
f(x) == acos(-sqrt(C1/cos(x)**2)), f(x) == acos(sqrt(C1/cos(x)**2))]
```

```
>>> dsolve(eq, hint='best')
f(x) == C1/(C2*x - 1)
```

classify_ode()

sympy.solvers.ode.**classify_ode**(*eq*, *func=None*, *dict=False*, ***kwargs*)

Returns a tuple of possible dsolve() (page 1257) classifications for an ODE.

The tuple is ordered so that first item is the classification that dsolve() (page 1257) 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, thought there are always exceptions. To make dsolve() (page 1257) use a different classification, use `dsolve(ODE, func, hint=<classification>)`. See also the dsolve() (page 1257) docstring for different meta-hints you can use.

If `dict` is true, classify_ode() (page 1259) will return a dictionary of `hint:match` expression terms. This is intended for internal use by dsolve() (page 1257). 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 `hintname` is the name of the hint without `_Integral`.

See allhints (page 1263) or the ode (page 1284) docstring for a list of all supported hints that can be returned from classify_ode() (page 1259).

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 529) class in it. Note that a hint may do this anyway if integrate() (page 109) 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 109) is an expensive routine. If dsolve() (page 1257) hangs, it is probably because integrate() (page 109) 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, nth 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 529)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',
'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 135) that the solution satisfies, e.g. `Eq(f(x), C1), Eq(f(x) + C1, 0)`; or simply an `Expr` (page 97), 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, ..., nth 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`. Note that sometimes this function will `False`, but with an expression that is identically equal to `0`, instead of returning `True`. This is because `simplify()` (page 1182) cannot reduce the expression to `0`. If an expression returned by this function vanishes identically, then `sol` really is a solution to `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 $f(x, y, \cdots)$ is homogeneous of order n if $f(tx, ty, \cdots) = t^n f(x, y, \cdots)$.

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, **kwargs*)

The functions xi and eta, are called the infinitesimals which help in the process of finding a new co-ordinate system, in which a differential equation can be simplified. They are tangents to the coordinate curves, in which the differential equation is simplified.

Consider the transformation (x, y) –> (X, Y) such that X and Y are f(x, y, lambda) and g(x, y, lambda) and such that the differential equation remains invariant. Xi and eta are the tangents to the transformed coordinates X and Y, when lambda is the identity.

The infinitesimals can be found by solving the following Partial Differential Equation.

```
>>> 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)
>>> pprint(genform)
/d              d           \                     d              2       d
|--(eta(x, y)) - --(xi(x, y))|*h(x, y) - eta(x, y)*--(h(x, y)) - h (x, y)*--(x
\dy             dx          /                     dy                      dy

                   d             d
i(x, y)) - xi(x, y)*--(h(x, y)) + --(eta(x, y)) = 0
                   dx            dx
```

Once the infinitesimals are found, the following partial differential equations would help us find the transformed coordinates (X, Y)

1. X.diff(x)*xi + X.diff(y)*eta = 0
2. Y.diff(x)*xi + Y.diff(y)*eta = 1

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},
{eta(x, f(x)): f(x), xi(x, f(x)): 0},
{eta(x, f(x)): 0, xi(x, f(x)): x**(-2)},
{eta(x, f(x)): x**2*f(x) + f(x), xi(x, f(x)): 1}]
```

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 for the given first order differential equation. As of now, it simply checks, by substituting the infinitesimals in the partial differential equation. (eta.diff(x) + (eta.diff(y) - xi.diff(x))*h - (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y))) = 0 where eta, and xi are the infinitesimals and h(x,y) = 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 zero.

5.28.2 Hint Functions

These functions are intended for internal use by `dsolve()` (page 1257) and others. Unlike User Functions (page 1256), above, these are not intended for every-day use by ordinary SymPy users. Instead, functions such as `dsolve()` (page 1257) 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', 'Riccati_special_r**

This is a list of hints in the order that they should be preferred by `classify_ode()` (page 1259). 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 1257) 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**(**args, **kwargs*)

Simplifies ODEs, including trying to solve for `func` and running `constantsimp()` (page 1265).

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 529)s in the expression, if the hint is not an `_Integral` hint.

This function should have no effect on expressions returned by `dsolve()` (page 1257), as `dsolve()` (page 1257) already calls `odesimp()` (page 1264), but the individual hint functions do not call `odesimp()` (page 1264) (because the `dsolve()` (page 1257) 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
                       ----
                       f(x)
                         /
                        |
                        |   /        1   \
                        |  -|u2 + -------|
                        |   |        /1 \|
                        |   |     sin|--||
                        |   \        \u2//
log(f(x)) = log(C1) +   |  ---------------- d(u2)
                        |          2
                        |        u2
                        |
                       /

>>> pprint(odesimp(eq, f(x), 1,
... hint='1st_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.

This is a simple function that goes through and renumbers any `Symbol` (page 117) 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 1265).

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
>>> 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**(**args, **kwargs*)

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with `dsolve()` (page 1257), and is not intended for general use.

Simplification is done by "absorbing" the arbitrary constants in to 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, `C1`, `C2`, `C3`. 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 is done with limited assistance: terms of `Add` (page 131)s are collected to try join constants and powers with exponents that are `Add` (page 131)s are expanded so $e^x(C_1 \cos(x) + C_2 \cos(x))$ will simplify to $e^x C_1 \cos(x)$ and e^{C_1+x} will be simplified to $C_1 e^x$.

Use `constant_renumber()` (page 1265) 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 1257).

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, x, 3)
C1*x
>>> constantsimp(C1 + 2 + x + y, x, 3)
C1 + x
>>> constantsimp(C1*C2 + 2 + x + y + C3*x, x, 3)
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 529)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
sol solved for func	-2
sol not solved for func but can be	-1
sol is not solved nor solvable for func	len(str(sol))
sol contains an Integral (page 529)	oo

oo here means the SymPy infinity, which should compare greater than any integer.

If you already know solve() (page 1295) cannot solve sol, you can use trysolving=False to skip that step, which is the only potentially slow step. For example, dsolve() (page 1257) 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]]
[26, 33]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
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))
            x                y
            /                /
           |                |
F(x, y) =  |  P(t, y) dt +  |  Q(x0, t) dt = C1
           |                |
          /                /
          x0               y0
```

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='1st_exact')
x*cos(f(x)) + f(x)**3/3 == C1
```

1st_homogeneous_coeff_best

sympy.solvers.ode.**ode_1st_homogeneous_coeff_best**(*eq, func, order, match*)

Returns the best solution to an ODE from the two hints `1st_homogeneous_coeff_subs_dep_div_indep` and `1st_homogeneous_coeff_subs_indep_div_dep`.

This is as determined by `ode_sol_simplicity()`.

See the `ode_1st_homogeneous_coeff_subs_indep_div_dep()` and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` docstrings for more information on these hints. Note that there is no `1st_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='1st_homogeneous_coeff_best', simplify=False))
                        /    2    \
                        | 3*x     |
                     log|----- + 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 1261).

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)
log(x) = C1 +    |  ---------------- d(u1)
                 |  u1*h(u1) + g(u1)
                 |
                /
```

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()` and `ode_1st_homogeneous_coeff_subs_indep_div_dep()`.

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))
                       /          3   \
                       |3*f(x)   f (x)|
                    log|------ + -----|
                       |  x         3 |
                       \           x  /
log(x) = log(C1) - -------------------
                            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 $u_2 = \frac{\text{<independent variable>}}{\text{<dependent variable>}}$.

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 1261).

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)
 / x  \    / x  \ d
g|----| + h|----|*--(f(x))
 \f(x)/    \f(x)/ dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'))
             x
            ----
            f(x)
              /
             |
             |       -g(u2)
             |  ---------------- d(u2)
             |  u2*g(u2) + h(u2)
             |
            /

f(x) = C1*e
```

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 1268) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()`.

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))
                         /    2    \
                         | 3*x     |
                      log|----- + 1|
                         | 2       |
                         \f (x)    /
log(f(x)) = log(C1) - --------------
                            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)
             d
P(x)*f(x) + --(f(x)) = Q(x)
             dx
>>> pprint(dsolve(genform, f(x), hint='1st_linear_Integral'))
       /       /                    \
       |      |                     |
       |      |         /           |     /
       |      |        |            |    |
       |      |        | P(x) dx    |  - | P(x) dx
       |      |        |            |    |
       |      |       /             |   /
f(x) = |C1 +  | Q(x)*e           dx|*e
       |      |                     |
       \     /                      /
```

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 1272)). 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)
            d                 n
P(x)*f(x) + --(f(x)) = Q(x)*f (x)
            dx
>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'))
                                                                                  1
                                                                                ----
                                                                               1 - n
       //                /                            \                     \
       ||               |                             |                     |
       ||               |                   /         |             /       |
       ||               |                 |           |           |         |
       ||               |        (1 - n)* | P(x) dx   |  (-1 + n)* | P(x) dx|
       ||               |                 |           |           |         |
       ||               |                /            |          /          |
f(x) = ||C1 + (-1 + n)* | -Q(x)*e                   dx|*e                   |
       ||               |                             |                     |
       \\               /                             /                     /
```

Note that the equation is separable when $n = 1$ (see the docstring of `ode_separable()` (page 1279)).

```
>>> 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'))
                1
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)
                  2                    2
        /d       \          d          d
g(f(x))*|--(f(x))|  + h(x)*--(f(x)) + ---(f(x)) = 0
        \dx      /          dx           2
                                       dx
>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))
                                    f(x)
          /                      /
         |                      |
         |     /                |     /
         |    |                 |    |
         |  - | h(x) dx         |    | g(y) dy
         |    |                 |    |
         |   /                  |   /
C1 + C2* | e            dx +    |  e           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) = -\/ C1 + C2*log(x) , f(x) = \/ 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)
        /                                /        ________________        \\
        |           ________________     |       /                2        ||
        |          /                2     |     \/  4*b*d - (a + c)  *log(x)||
       -|a + c - \/  4*b*d - (a + c)  *tan|C1 + ----------------------------||
        \                                \                 2*a             //
f(x) = ------------------------------------------------------------------------
                                       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

nth_linear_constant_coeff_homogeneous

sympy.solvers.ode.**ode_nth_linear_constant_coeff_homogeneous**(*eq, func, order, match, returns='sol'*)

Solves an nth 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 $eIx = \cos(x) + I\sin(x)$. Complex roots always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as $e^{ax}\left(C_1 \cos(bx) + C_2 \sin(bx)\right)$.

If SymPy cannot find exact roots to the characteristic equation, a `RootOf` (page 1004) 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')
...
f(x) == C1*exp(x*RootOf(_x**5 + 10*_x - 2, 0)) +
C2*exp(x*RootOf(_x**5 + 10*_x - 2, 1)) +
C3*exp(x*RootOf(_x**5 + 10*_x - 2, 2)) +
C4*exp(x*RootOf(_x**5 + 10*_x - 2, 3)) +
C5*exp(x*RootOf(_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, order, match*)

Solves an nth 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'))
       /            4\
       |           x |  -x   4*sin(2*x)   3*cos(2*x)
f(x) = |C1 + C2*x + --|*e   - ---------- + ----------
       \           3 /           25           25
```

nth_linear_constant_coeff_variation_of_parameters

sympy.solvers.ode.**ode_nth_linear_constant_coeff_variation_of_parameters**(*eq, func, order, match*)

Solves an nth order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{x=1}^{n} c_i(x) y_i(x),$$

where y_i is the ith solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by

$$\sum_{x=1}^{n} \left(\int \frac{W_i(x)}{W(x)}\, dx \right) y_i(x),$$

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 ith column replaced with $[0, 0, \cdots, 0, P(x)]$.

This method is general enough to solve any nth 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 1257) 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/encyclopedia/VariationOfParameters.html
- 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'))
       /                     3                \
       |                 2   x *(6*log(x) - 11)|  x
f(x) = |C1 + C2*x + C3*x  + ------------------|*e
       \                            36        /
```

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.separatevars()` 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()` 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)
             d
a(x)*b(f(x))*--(f(x)) = c(x)*d(f(x))
             dx
>>> pprint(dsolve(genform, f(x), hint='separable_Integral'))
     f(x)
   /                  /
  |                  |
  |  b(y)            | c(x)
  |  ---- dy = C1 +  | ---- dx
  |  d(y)            | a(x)
```

```
  |                   |
 /                   /
```

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*g(x)                  \\
       |     ||   --------     for k(x) = 0||
       |     ||     f(x)                   ||  -y*k(x)
       |     ||                            ||  --------
       |     ||       y*k(x)               ||    f(x)
l(y) = |C1 + |<       ------               ||*e
       |     ||        f(x)                ||
       |     ||-g(x)*e                     ||
       |     ||--------------   otherwise  ||
```

```
       |     ||     k(x)                    ||
       \     \\                             //
```

See Also:

sympy.solvers.ode.ode_1st_linear() (page 1272)

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')
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

$$y' + F\left(\frac{a_1 x + b_1 y + c_1}{a_2 x + b_2 y + c_2}\right) = 0,$$

where a_1, b_1, c_1, a_2, b_2, c_2 are constants and $a_1 b_2 - a_2 b_1 \neq 0$.

This can be solved by substituting:

$$x = x' + \frac{b_2 c_1 - b_1 c_2}{a_2 b_1 - a_1 b_2}$$
$$y = y' + \frac{a_1 c_2 - a_2 c_1}{a_2 b_1 - a_1 b_2}.$$

This substitution reduces the equation to a homogeneous differential equation.

See Also:

sympy.solvers.ode.ode_1st_homogeneous_coeff_best() (page 1268), sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep() (page 1270), sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep() (page 1269)

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')
>>> df = f(x).diff(x)
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[f(x) == -x - sqrt(C1 + 7*x**2) - 1, f(x) == -x + sqrt(C1 + 7*x**2) - 1]
>>> pprint(dsolve(eq, hint='linear_coefficients'))
                ___________                    ___________
               /         2                    /         2
[f(x) = -x - \/  C1 + 7*x   - 1, f(x) = -x + \/  C1 + 7*x   - 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)
                  / n     \
d          f(x)*g\x *f(x)/
--(f(x)) + ---------------
dx                 x
>>> pprint(dsolve(genform, hint='separable_reduced'))
 n
x *f(x)
  /
 |
 |         1
 |    ------------ dy = C1 + log(x)
 |    y*(n - g(y))
 |
 /
```

See Also:

sympy.solvers.ode.ode_separable() (page 1279)

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')
[f(x) == (-sqrt(C1*x**2 + 1) + 1)/x, f(x) == (sqrt(C1*x**2 + 1) + 1)/x]
>>> pprint(dsolve(eq, hint='separable_reduced'))
            ___________                ___________
           /     2                    /     2
        - \/  C1*x  + 1  + 1         \/  C1*x  + 1  + 1
[f(x) = --------------------, f(x) = ------------------]
                 x                           x
```

lie_group

1st_power_series

2nd_power_series_ordinary

2nd_power_series_regular

5.28.3 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

abaco1_product

bivariate

chi

abaco2_similar

function_sum

abaco2_unique_unknown

abaco2_unique_general

linear

5.28.4 Information on the ode module

This module contains `dsolve()` (page 1257) and different helper functions that it uses.

`dsolve()` (page 1257) 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 1257).

Functions in this module

These are the user functions in this module:

- `dsolve()` (page 1257) - Solves ODEs.
- `classify_ode()` (page 1259) - Classifies ODEs into possible hints for `dsolve()` (page 1257).
- `checkodesol()` (page 1260) - Checks if an equation is the solution to an ODE.
- `homogeneous_order()` (page 1261) - Returns the homogeneous order of an expression.

These are the non-solver helper functions that are for internal use. The user should use the various options to `dsolve()` (page 1257) to obtain the functionality provided by these functions:

- `odesimp()` (page 1264) - Does all forms of ODE simplification.
- `ode_sol_simplicity()` (page 1266) - A key function for comparing solutions by simplicity.
- `constantsimp()` (page 1265) - Simplifies arbitrary constants.
- `constant_renumber()` (page 1265) - 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.
- 2nd order Liouville differential equations.
- nth order linear homogeneous differential equation with constant coefficients.
- nth order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.
- nth 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 1259) 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 1259) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 529) class. `dsolve()` (page 1257), which is the user wrapper function around all of this, will then call `odesimp()` (page 1264) 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 1257) 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 1284) module will become more robust, and unhindered by special case hacks. WolphramAlpha 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 nth 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 1268), 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 1257) 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 1257) code. See also the classify_ode() (page 1259) 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 1263) 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 1257). 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 1259) (if the match function is more than just a few lines, like _undetermined_coefficients_match(), it should go outside of classify_ode() (page 1259)). It should match the ODE without solving for it as much as possible, so that classify_ode() (page 1259) 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 1259). classify_ode() (page 1259) will then send this to dsolve() (page 1257), 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 1259) 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 1257) 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 C.Integral() instead of integrate() (page 109). 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 88) 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 1264) 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 1264) for it. For example, solutions returned from the 1st_homogeneous_coeff hints often have many log() terms, so odesimp() (page 1264) calls logcombine() 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 1265) take better advantage of it. It is better to put simplification in odesimp() (page 1264) than in your method, because it can then be turned off with the simplify flag in dsolve() (page 1257). 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 1260) (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 1257), 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 nth 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.

5.29 PDE

5.29.1 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`, `pde_separate_mul`

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, x)/X(x), Derivative(T(t), t, t)/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, x)/X(x), Derivative(Y(y), y, y)/Y(y)]
```

pdsolve()

sympy.solvers.pde.**pdsolve**(*eq, func=None, hint='default', dict=False, solvefun=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)
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 at the near the beginning of the list will produce better solutions faster than those near the end, thought 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
```

5.29.2 Hint Methods

These funcions 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*f(x,y).diff(x) + b*f(x,y).diff(y) + c*f(x,y) = 0 where a, b and c are constants.

The general solution of the differential equation, can be found by the method of characteristics. It is given by f(x,y) = F(b*x - a*y)*exp(-c/(a**2 + b**2)*(a*x + b*y))

```
>>> 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))
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, solvefun*)

Solves a first order linear partial differential equation with constant coefficients.

The general form of this partial differential equation is a*f(x,y).diff(x) + b*f(x,y).diff(y) + c*f(x,y) = G(x,y) where a, b and c are constants and G can be an arbitrary function in x and y.

The general solution of the PDE is

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> G = Function('G')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*u + b*ux + c*uy - G(x,y)
>>> pprint(genform)
              d               d
a*f(x, y) + b*--(f(x, y)) + c*--(f(x, y)) - G(x, y)
              dx              dy
>>> pprint(pdsolve(genform, hint='1st_linear_constant_coeff_Integral'))
          //          b*x + c*y                                                \
          ||              /                                                    |
          ||             |                                                     |
          ||             |                                       a*xi          |
          ||             |                                     -------         |
          ||             |                                      2    2         |
          ||             |      /b*xi + c*eta  -b*eta + c*xi\  b  + c         |
          ||             |     G|------------, -------------|*e        d(xi)|
          ||             |      |   2    2         2    2   |                 |
          ||             |      \  b  + c         b  + c    /                 |
          ||             |                                                     |
          ||            /                                                      |
          ||                                                                   |
f(x, y) = ||F(eta) + -----------------------------------------------------|*
          ||                                 2    2                           |
          \\                               b  + c                            /

        \|
        ||
        ||
        ||
        ||
        ||
        ||
        ||
        ||
```

```
  -a*xi ||
 -------||
  2    2||
 b  + c ||
e       ||
        ||
        /|eta=-b*y + c*x, xi=b*x + c*y
```

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 = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
f(x, y) == (F(4*x + 2*y) + exp(x/2 + 4*y)/15)*exp(x/2 - y)
```

pde_1st_linear_variable_coeff

5.29.3 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.

5.30 Solvers

The *solvers* module in SymPy implements methods for solving equations.

5.30.1 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:

- univariate polynomial,
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- sytems 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.

'warning=True (default is False)' show a warning if checksol() could not conclude.

'simplify=True (default)' simplify all but cubic and quartic solutions 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, ...

'minimal=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 minimal=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)

Notes

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 dicontinuity) 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 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)
And(im(x) == 0, re(x) < 3)
```

- to always get a list of solution mappings, use flag dict=True

```
>>> solve(x - 3, dict=True)
[{x: 3}]
>>> solve([x - 3, y - 1], dict=True)
[{x: 3, 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], set([(-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], set([(-y,), (y,)]))
>>> solve(x**4 - 1, x, set=True)
([x], set([(-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)], set([(-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)
[{x: y**2}]
>>> solve(x**2 - y, x, y)
[{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], set([(-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)
[{x: -y}, {x: y}]
>>> solve(x**2 - y**2/exp(x), x, y)
[{x: 2*LambertW(y/2)}]
>>> solve(x**2 - y**2/exp(x), y, x)
[{y: -x*exp(x/2)}, {y: x*exp(x/2)}]
```

• iterable of one or more of the above

- involving relationals or bools

```
>>> solve([x < 3, x - 2])
And(im(x) == 0, re(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], set([(-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)], set([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}
```

`sympy.solvers.solvers.`**`solve_linear`**(*lhs, rhs=0, symbols=*[ ], *exclude=*[ ])

Return a tuple derived from f = lhs - rhs that is either:

(numerator, denominator) of `f` If this comes back as (0, 1) it means that `f` is independent of the symbols in `symbols`, e.g:

```
y*cos(x)**2 + y*sin(x)**2 - y = y*(0) = 0
cos(x)**2 + sin(x)**2 = 1
```

If it comes back as (0, 0) there is no solution to the equation amongst the symbols given.

If the numerator is not zero then the function is guaranteed to be dependent on a symbol in `symbols`.

or

(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 and mul=True expansion, so the solution will correspond strictly to a unique solution.

Examples

```
>>> from sympy.solvers.solvers import solve_linear
>>> from sympy.abc import x, y, z
```

These are linear in x and 1/x:

```
>>> 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 is a symbol then (0, 0) is returned if the solution for that symbol would have set any denominator to 0:

```
>>> solve_linear(1/(1/x - 2))
(0, 0)
>>> 1/(1/x) # to SymPy, this looks like x ...
x
>>> solve_linear(1/(1/x)) # so a solution is given
(x, 0)
```

If x is allowed to cancel, then this appears linear, but this sort of cancellation is not done so the solution will always satisfy the original expression without causing a division by zero error.

```
>>> solve_linear(x**2*(1/x - z**2/x))
(x**2*(-z**2 + 1), x)
```

You can give a list of what you prefer for x candidates:

```
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

You can also indicate what variables you don't want to consider:

```
>>> solve_linear(x + y + z, exclude=[x, z])
(y, -x - z)
```

If only x was excluded then a solution for y or z might be obtained.

`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 Nx(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, ..., 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.**tsolve**(*eq, sym*)

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.

Overdetermined systems are supported.

```
>>> from sympy import Symbol, nsolve
>>> import sympy
>>> sympy.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))
[-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
```

mpmath.findroot is used, you can find there more extensive documentation, especially concerning keyword parameters and available solvers.

sympy.solvers.solvers.**check_assumptions**(*expr, **assumptions*)

Checks whether expression $expr$ satisfies all assumptions.

$assumptions$ is a dict of assumptions: {'assumption': True|False, ...}.

Examples

```
>>> from sympy import Symbol, pi, I, exp
>>> from sympy.solvers.solvers import 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
```

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. `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.

5.30.2 Ordinary Differential equations (ODEs)

See *ODE* (page 1256).

5.30.3 Partial Differential Equations (PDEs)

See *PDE* (page 1287).

5.30.4 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
```

5.30.5 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) + ... + a_{0}(n) y(n) = f

where a_{i}(n), for i=0..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..m, maps to y(n_i).

As an example lets consider the following recurrence:

(n - 1) y(n + 2) - (n**2 + 3 n - 2) y(n + 1) + 2 n (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.**rsolve_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) == m*n**(m-1) recurrence, which has solution b(n) = B_m + C. For example:

```
>>> from sympy import Symbol, rsolve_poly
>>> n = Symbol('n', integer=True)

>>> rsolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

For more information on implemented algorithms refer to:

[1] 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.

[2] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.

[3] M. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996.

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.

For more information on the implemented algorithm refer to:

[1] 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

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

For more information on the implemented algorithm refer to:

[1] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.

[2] M. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996.

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
```

5.30.6 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)]
```

5.30.7 Diophantine Equations (DEs)

See *Diophantine* (page 1309)

5.31 Diophantine

5.31.1 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, $f(x_1, x_2, \ldots x_n) = 0$ where $n \geq 2$ and $x_1, x_2, \ldots x_n$ are integer variables. If we can find n integers $a_1, a_2, \ldots a_n$ such that $x_1 = a_1, x_2 = a_2, \ldots x_n = a_n$ satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in [3] and [4].

Currently, following five types of Diophantine equations can be solved using `diophantine()` and other helper functions of the Diophantine module.

- Linear Diophantine equations: $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$.
- General binary quadratic equation: $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation: $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$
- Extended Pythagorean equation: $a_1x_1^2 + a_2x_2^2 + \ldots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
- General sum of squares: $x_1^2 + x_2^2 + \ldots + x_n^2 = k$

5.31.2 Module structure

This module contains `diophantine()` and helper functions that are needed to solve certain Diophantine equations. It's structured in the following manner.

- `diophantine()`
  - `diop_solve()`
    - `classify_diop()`

[3] Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations
[4] Diophantine Equation, Wolfram Mathworld, [online]. Available: http://mathworld.wolfram.com/DiophantineEquation.html

```
        * diop_linear()
        * diop_quadratic()
        * diop_ternary_quadratic()
        * diop_general_pythagorean()
        * diop_general_sum_of_squares()
    - merge_solution()
```

When an equation is given to `diophantine()`, it factors the equation(if possible) and solves the equation given by each factor by calling `diop_solve()` separately. Then all the results are combined using `merge_solution()`.

`diop_solve()` internally uses `classify_diop()` 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()` returned "linear" as the type of the equation, then `diop_solve()` calls `diop_linear()` to solve the equation.

Each of the functions, `diop_linear()`, `diop_quadratic()`, `diop_ternary_quadratic()`, `diop_general_pythagorean()` and `diop_general_sum_of_squares()` 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.

5.31.3 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)
set([(3*t - 5, -2*t + 5)])
```

Note that stepping one more level below the highest API, we can solve the very same equation by calling `diop_solve()`.

```
>>> from sympy.solvers.diophantine import diop_solve
>>> diop_solve(2*x + 3*y - 5)
(3*t - 5, -2*t + 5)
```

Note that it returns a tuple rather than a set. `diophantine()` always return a set of tuples. But `diop_solve()` 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()`, which is what `diop_solve()` calls internally.

```
>>> from sympy.solvers.diophantine import diop_linear
>>> diop_linear(2*x + 3*y - 5)
(3*t - 5, -2*t + 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 - 5, -2*m + 5)
```

Please note that for the moment, user 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 refer [5] and [6] 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)
set([(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)
set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])
>>> diophantine(x**2 + 2*x*y + y**2 - 3*x - 3*y)
set([(t, -t), (t, -t + 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 infinte 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 expand
>>> n = symbols("n", Integer=True)
```

[5] Methods to solve Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,[online], Available: http://www.alpertron.com.ar/METHODS.HTM

[6] Solving the equation ax^2+ bxy + cy^2 + dx + ey + f= 0, [online], Available: http://www.jpr2718.org/ax2p.pdf

```
>>> s = diophantine(x**2 -  2*y**2 - 2*x - 4*y, n)
>>> x_n, y_n = s.pop()
>>> expand(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) +
>>> expand(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) +
```

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)
```

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()`

```
>>> 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 belows.

```
>>> 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()`

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation does not have solutions.

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)
set()
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
set([(-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 about a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use `diop_ternary_quadratic()`.

```
>>> 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()` first converts the given equation to an equivalent equation of the form $w^2 = AX^2 + BY^2$ and then it uses `descent()` 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()`.

```
>>> 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, $a_1x_1^2 + a_2x_2^2 + \ldots + a_nx_n^2 = a_{n+1}x_{n+1}^2$ and the general sum of squares equation, $x_1^2 + x_2^2 + \ldots + x_n^2 = k$ 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)
set([(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*t
```

function `diop_general_pythagorean()` can also be called directly to solve the same equation. This is true about the general sum of squares too. Either you can call `diop_general_pythagorean()` or use the high level API.

```
>>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)
set([(8, 4, 4, 4, 0, 0)])
```

If you want to get a more thorough idea about the the Diophantine module please refer to the following blog.

http://thilinaatsympy.wordpress.com/

5.31.4 References

5.31.5 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions are intended for use by ordinary users of SymPy.

diophantine()

diop_solve()

classify_diop()

diop_linear()

base_solution_linear()

diop_quadratic()

diop_DN()

cornacchia()

diop_bf_DN()

transformation_to_DN()

find_DN()

diop_ternary_quadratic()

square_factor()

descent()

diop_general_pythagorean()

diop_general_sum_of_squares()

partition()

sum_of_three_squares()

sum_of_four_squares()

5.31.6 Internal Functions

These functions are intended for the internal use in Diophantine module.

merge_solution

divisible

extended_euclid

PQa

equivalent

simplified

parametrize_ternary_quadratic

diop_ternary_quadratic_normal

ldescent

gaussian_reduce

holzer

prime_as_sum_of_two_squares

pairwise_prime

make_prime

reconstruct

5.32 Tensor Module

A module to manipulate symbolic objects with indices including tensors

5.32.1 Contents

Indexed Objects

Module that defines indexed objects

The classes IndexedBase, Indexed and Idx would represent a matrix element M[i, j] as in the following graph:

```
1) The Indexed class represents the entire indexed object.
           |
       ,___|___,
        M[i, j]
       /   \__\______
       |             |
       |             |
       |     2) The Idx class represent indices and 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.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> 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
(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])
(set([i]), {})
>>> get_contraction_structure(A[i, j, j])
{(j,): set([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: the Idx constructor is rather pedantic in that it only accepts integer arguments. The only exception is that you can use oo and -oo to specify an unbounded range. For all other cases, both label and bounds must be declared as integers, e.g. if n is given as an argument then n.is_integer must return True.

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.tensor import IndexedBase, Idx
>>> from sympy import 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)
```

The label can be a literal integer instead of a string/Symbol:

```
>>> idx = Idx(2, n); idx.lower, idx.upper
(0, n - 1)
>>> idx.label
2
```

label

Returns the label (Integer or integer expression) of the Idx object.

Examples

```
>>> from sympy import Idx, Symbol
>>> Idx(2).label
2
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

lower

Returns the lower bound of the Index.

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 Index.

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.tensor import Indexed, IndexedBase, Idx
>>> from sympy import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects are 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.tensor import Indexed, IndexedBase, Idx
>>> from sympy import 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.tensor import Indexed, Idx
>>> from sympy import 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.tensor import Indexed, Idx
>>> from sympy import 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.tensor.indexed import IndexedBase, Idx
>>> from sympy import 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)
```

args

Returns the arguments used to create this IndexedBase object.

Examples

```
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).args
(A, (x, y))
```

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
```

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)
```

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 stucture 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, ommitting dicts with None as the one and only key.

Note: The presence of expressions among the dictinary 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,): set([x[i]*y[i]]), (j,): set([A[j, j]])}
>>> get_contraction_structure(x[i]*y[j])
{None: set([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]
set([x[i, i]*y[j, j]])
```

Factors are contracted "first":

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{(i,): set([x[i, i]])}, {(j,): set([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)
[x[i]*(y[i] + A[i, j]*x[j]), (i,)]
>>> d[(i,)]
set([x[i]*(y[i] + A[i, j]*x[j])])
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[{None: set([y[i]]), (j,): set([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]
set([A[j, j]**A[i, i]])
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): set([A[j, j]])}
>>> nested_contractions[1]
{(i,): set([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 contactions 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])
(set([i]), {})
```

Exceptions

An IndexConformanceException means that the terms ar 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])
> (set([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 corrispondence 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, TensorManag
>>> 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)
```

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`		
`dim_eps`		
`dummy_fmt`		

class `sympy.tensor.tensor.`**`TensorIndex`**

Represents an abstract tensor index.

Parameters **name** : name of the index

tensortype : `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 `tensortype.dummy_fmt`

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorIndex, TensorSymmetry, TensorType, ge
>>> 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)
```

Attributes

name	
tensortype	
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 : list of 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 260)

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, tensor_indices, TensorSymmetry, TensorType,
>>> 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	

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, tensorsymmetry, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1]*2)
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> A = S2('A')
```

Attributes

name		
index_types		
rank		
types	equal to typ.types	
symmetry	equal to typ.symmetry	
comm	commutation group	

commutes_with(*other*)

Returns 0 (1) if self and other (anti)commute.

Returns None if self and other do not (anti)commute.

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 monoterm 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.

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)
```

Attributes

`args`	tuple of addends	
`rank`	rank of the tensor	
`free_args`	list of the free indices in sorted order	

`canon_bp`()

canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

`contract_metric`(*g, contract_all=False*)

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_args()

Returns `(g, dummies, msym, v)`, the entries of `canonicalize`

see `canonicalize` in `tensor_can.py`

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, contract_all=False*)

Raise or lower indices with the metric `g`

`g` metric

`contract_all` if True, eliminate all `g` which are contracted

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)
```

static from_indices(**indices*)

Convert `indices` into `free`, `dum` for single component tensor

free list of tuples (index, pos, 0), where `pos` is the position of index in the list of indices formed by the component tensors

`dum` list of tuples `(pos_contr, pos_cov, 0, 0)`

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, TensMul
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2, m3 = tensor_indices('m0,m1,m2,m3', Lorentz)
>>> TensMul.from_indices(m0, m1, -m1, m3)
([(m0, 0, 0), (m3, 3, 0)], [(1, 2, 0, 0)])
```

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_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]
```

perm2tensor(*g, canon_bp=False*)

Returns the tensor corresponding to the permutation `g`

`g` permutation corrisponding to the tensor in the representation used in canonicalization

`canon_bp` if True, then `g` is the permutation corresponding to the canonical form of the tensor

sorted_components()

Returns a tensor with sorted components

The sorting is done taking into account the commutation group of the component tensors.

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)]
```

substitute_indices(**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.substitute_indices((i,j), (j, k))
A(j, L_0)*B(-L_0, -k)
```

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) -> 2/3*R(m,n,p,q) - 1/3*R(m,q,n,p) + 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, 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
```

5.33 Utilities

This module contains some general purpose utilities that are used across SymPy.

Contents:

5.33.1 Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the *Tensor Module* (page 1315). 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
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**(*generator*, *filepath=None*, *flags=[]*, *verbose=False*)

Wrapper that uses Cython

dump_pyx(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

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 basename of the prefix is used.

empty Optional. When True, empty lines are included to structure the source files. [DEFAULT=True]

class sympy.utilities.autowrap.**DummyWrapper**(*generator*, *filepath=None*, *flags=[]*, *verbose=False*)

Class used for testing independent of backends

class sympy.utilities.autowrap.**F2PyCodeWrapper**(*generator*, *filepath=None*, *flags=[]*, *verbose=False*)

Wrapper that uses f2py

sympy.utilities.autowrap.**autowrap**(*expr, language='F95', backend='f2py', tempdir=None, args=None, flags=[], verbose=False, helpers=[]*)

Generates python callable binaries based on the math expression.

expr The SymPy expression that should be wrapped as a binary routine

Optional arguments

language The programming language to use, currently 'C' or 'F95'

backend The wrapper backend to use, currently f2py or Cython

tempdir 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 Sequence of the formal parameters of the generated code, if ommited the function signature is determined by the code generator.

flags Additional option flags that will be passed to the backend

verbose If True, autowrap will not mute the command line backends. This can be helpful for debugging.

helpers Used to define auxillary expressions needed for the main expr. If the main expression need to do call a specialized function it should be put in the `helpers` list. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<funtion_name>, <sympy_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

```
>>> 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().

```
>>> 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)
<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, **kwargs*)

Generates a binary ufunc-like lambda function for numpy arrays

args Either a Symbol or a tuple of symbols. Specifies the argument sequence for the ufunc-like function.

expr A SymPy expression that defines the element wise operation

kwargs Optional keyword arguments are forwarded to autowrap().

The returned function can only act on one array at a time, as only the first argument accept arrays as input.

Note: a *proper* numpy ufunc is required to support broadcasting, type casting and more. The function returned here, may not qualify for numpy's definition of a ufunc. That why we use the term ufunc-like.

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)
>>> f([1, 2, 3], 2)
[ 3.  6.  11.]
>>> a = f(np.arange(5), 3)
>>> isinstance(a, np.ndarray)
True
>>> print a
[ 3. 4. 7. 12. 19.]
```

5.33.2 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
```

Impementation 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 and python 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
- 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
- ...

class `sympy.utilities.codegen.`**`Routine`**(*name*, *expr*, *argument_sequence=None*)

Generic description of an evaluation routine for a set of sympy expressions.

A CodeGen class can translate instances of this class into C/Fortran/... code. 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 ot the list.

`variables`

Returns a set containing all variables possibly used in this 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*, *pyname*)

Holds strings for a certain datatype in different programming languages.

sympy.utilities.codegen.**get_default_datatype**(*expr*)
Derives a decent data type based on the assumptions 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.**Result**(*expr*, *datatype=None*, *precision=None*)
An expression for a scalar 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.

class sympy.utilities.codegen.**CodeGen**(*project='project'*)
Abstract class for the code generators.

dump_code(*routines*, *f*, *prefix*, *header=True*, *empty=True*)
Write the code file by calling language specific methods in correct order

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Arguments

routines A list of Routine instances

f A file-like object to write the file to

prefix The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

Optional arguments

header When True, a header comment is included on top of each source file. [DEFAULT=True]

empty When True, empty lines are included to structure the source files. [DEFAULT=True]

write(*routines*, *prefix*, *to_files=False*, *header=True*, *empty=True*)
Writes all the source code files for the given routines.

The generate source is returned as a list of (filename, contents) tuples, or is written to files (see options). Each filename consists of the given prefix, appended with an appropriate extension.

routines A list of Routine instances to be written

prefix The prefix for the output files

to_files When True, the output is effectively written to files. [DEFAULT=False] Otherwise, a list of (filename, contents) tuples is returned.

header When True, a header comment is included on top of each source file. [DEFAULT=True]

empty When True, empty lines are included to structure the source files. [DEFAULT=True]

class sympy.utilities.codegen.**CCodeGen**(*project='project'*)
Generator for C code

The .write() method inherited from CodeGen will output a code file and an inteface file, <prefix>.c and <prefix>.h respectively.

dump_c(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code file by calling language specific methods in correct order

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Arguments

routines A list of Routine instances

f A file-like object to write the file to

prefix The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

Optional arguments

header When True, a header comment is included on top of each source file. [DEFAULT=True]

empty 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.

Arguments

routines A list of Routine instances

f A file-like object to write the file to

prefix The filename prefix, used to construct the include guards.

Optional arguments

header When True, a header comment is included on top of each source file. [DEFAULT=True]

empty When True, empty lines are included to structure the source files. [DEFAULT=True]

get_prototype(*routine*)

Returns a string for the function prototype for the given 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'*)

Generator for Fortran 95 code

The .write() method inherited from CodeGen will output a code file and an inteface file, <prefix>.f90 and <prefix>.h respectively.

dump_f95(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code file by calling language specific methods in correct order

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Arguments

routines A list of Routine instances

f A file-like object to write the file to

prefix The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

Optional arguments

header When True, a header comment is included on top of each source file. [DEFAULT=True]

empty 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.

Arguments

routines A list of Routine instances

f A file-like object to write the file to

prefix The filename prefix

Optional arguments

header When True, a header comment is included on top of each source file. [DEFAULT=True]

empty When True, empty lines are included to structure the source files. [DEFAULT=True]

get_interface(*routine*)

Returns a string for the function interface for the given routine and 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

sympy.utilities.codegen.**codegen**(*name_expr*, *language*, *prefix*, *project='project'*, *to_files=False*, *header=True*, *empty=True*, *argument_sequence=None*)

Write source code for the given expressions in the given language.

Mandatory Arguments

name_expr 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.

language A string that indicates the source code language. This is case insensitive. For the moment, only 'C' and 'F95' is supported.

prefix A prefix for the names of the files that contain the source code. Proper (language dependent) suffixes will be appended.

Optional Arguments

project A project name, used for making unique preprocessor instructions. [DEFAULT="project"]

to_files 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 When True, a header is written on top of each source file. [DEFAULT=True]

empty When True, empty lines are used to structure the code. [DEFAULT=True]

argument_sequence 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 aguments first, and then output or in-out arguments.

```
>>> 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), "C", "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) {
  return x + y*z;
}
>>> 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
```

5.33.3 Decorator

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 depenencies which need to be met for doctesting the docstrings of the decorated objects.

sympy.utilities.decorator.**no_attrs_in_subclass**

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.**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 1346) decorator allows threading over elements of `Add` class. If this behavior is not desirable use `xthreaded()` (page 1346) 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 1346) decorator disallows threading over elements of `Add` class. If this behavior is not desirable use `threaded()` (page 1346) decorator.

Functions using this decorator must have the following signature:

```
@xthreaded
def function(expr, *args, **kwargs):
```

5.33.4 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 partions.

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:
...         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
>>> 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(range(3))
[0, 1, 2]
>>> common_prefix(range(3), 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(range(3))
[0, 1, 2]
>>> common_suffix(range(3), 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.**flatten**(*iterable, levels=None, cls=None*)

Recursively denest iterable containers.

```
>>> from sympy.utilities.iterables import flatten

>>> flatten([1, 2, 3])
[1, 2, 3]
>>> 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*)

Generates the bell permutations of length n.

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 sympy.utilities.iterables import generate_bell
>>> from sympy import zeros, Matrix
>>> from sympy.core.compatibility import permutations
```

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), (1, 0, 2, 3), (1, 2, 0, 3), (1, 2, 3, 0), (2, 1, 3, 0)]
```

Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each bell permutations always has only two elements out of place relative to the previous permutation.

How the position of inversion varies across the elements can be seen by tracing out where the 0 appears in the permutations:

```
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
...     m[:, i] = Matrix(list(p))
>>> m.print_nonzero('X')
[ XXXXXX  XXXXXX  XXXXXX ]
[X XXXX XX XXXX XX XXXX X]
[XX XX XXXX XX XXXX XX XX]
[XXX  XXXXXX  XXXXXX  XXX]
```

`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`

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,
>>> 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,
```

sympy.utilities.iterables.**group**(*seq*, *multiple=True*)

Splits a sequence into a list of lists of equal, adjacent elements.

See Also:

multiset (page 1355)

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)
'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 1358), `multiset_partitions` (page 1355)

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:

```
>>> for p in kbins(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 which is either None (to give the simple partition of the 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 =', ordered
...     for p in kbins(range(3), 2, ordered=ordered):
```

```
...         print '    ', p
...
ordered = None
     [[0], [1, 2]]
     [[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 1351)

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 sympy.core.compatibility 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 1358), sympy.combinatorics.partitions.Partition (page 172), sympy.combinatorics.partitions.IntegerPartition (page 174), 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)
...     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)
set(['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, *args, **assumptions*)

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts.

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.`**`partitions`**(*n, m=None, k=None, size=False*)

Generate all partitions of integer n (>= 0).

Parameters **''m''** : integer (default gives partitions of all sizes)

limits number of parts in parition (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 172), `sympy.combinatorics.partitions.IntegerPartition` (page 174)

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 partion (the sum of the values in the returned dict) are limited with m:

```
>>> 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.`**`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).

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)]
```

Yields

subtree [sympy expression] All of the subtrees in the tree.

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 = 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(range(12), [2, [3], set([2]), (1, (3,), 1)])
[[0, 1, [2, 3, 4], set([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*)

Left rotates a list x by the number of steps specified in y.

Examples

```
>>> from sympy.utilities.iterables import rotate_right
>>> a = [0, 1, 2]
>>> 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*)

Sift the sequence, `seq` into a dictionary according to keyfunc.

OUTPUT: each element in expr is stored in a list keyed to the value of keyfunc for the element.

See Also:

`ordered`

Examples

```
>>> from sympy.utilities import sift
>>> from sympy.abc import x, y
>>> from sympy import sqrt, exp

>>> 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)]}
```

If you need to sort the sifted items it might be better to use `ordered` which can economically apply multiple sort keys to a squence while sorting.

`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 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]))
[(), (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:

```
+---+     +---+     +---+
| 7 |\    | 5 |     | 3 |
+---+ \   +---+     +---+
  |   _\___/ ____   _/ |
  |  /  \___/    \ /   |
  V  V           V V   |
 +----+         +---+  |
 | 11 |         | 8 |  |
 +----+         +---+  |
  | | \____   ___/ _   |
  | \      \ /    / \  |
  V  \     V V   /  V  V
+---+ \   +---+ |  +----+
| 2 |  |  | 9 | |  | 10 |
+---+  |  +---+ |  +----+
       \________/
```

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.`**`unrestricted_necklace`**(**args, **kwargs*)

Wrapper to necklaces to return a free (unrestricted) necklace.

`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)]
```

5.33.5 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 a sympy function, attach implementation to it.

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
>>> f = implemented_function(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.

```
>>> 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])'
```

sympy.utilities.lambdify.**lambdify**(*args, expr, modules=None, printer=None, use_imps=True*)

Returns a lambda function for fast calculation of numerical values.

If not specified differently by the user, SymPy functions are replaced as far as possible by either python-math, numpy (if available) or mpmath functions - exactly in this order. To change this behavior, the "modules" argument can be used. It accepts:

- the strings "math", "mpmath", "numpy", "sympy"
- 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.

Examples

```
>>> from sympy.utilities.lambdify import implemented_function, lambdify
>>> from sympy import sqrt, sin, Matrix
>>> from sympy import Function
>>> from sympy.abc import 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
>>> f = lambdify((x, y), Matrix((x, x + y)).T, modules='sympy')
>>> f(1, 2)
Matrix([[1, 3]])
```

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_imps` input parameter is False.

Usage

1. Use one of the provided modules:

   >> f = lambdify(x, sin(x), "math")

Attention: Functions that are not in the math module will throw a name error when the lambda function 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]

3. Use own dictionaries:

   >> def my_cool_function(x): ... >> dic = {"sin" : my_cool_function} >> f = lambdify(x, sin(x), dic)

   Now f would look like:

   >> lambda x: my_cool_function(x)

5.33.6 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 Pn0 = base_seq(0) cases XXX works only for m <= 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

5.33.7 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.`**`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.`**`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) # the \ appears as \ when printed
(
    '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'
    '    '
)
```

5.33.8 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 file(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_esource('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.

5.33.9 pytest

py.test 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`. The calling module must have `from __future__ import with_statement` as its first code line for compatibility with Python 2.5 in that case.

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
```

(Python 2.5's doctest cannot support with statements due to a bug in its __import__ handling. That's why the following examples are excluded from doctesting; the doctest: annotations can go once SymPy stops Python 2.5 support.)

```
>>> 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
```

5.33.10 Randomised Testing

Helpers for randomized testing

`sympy.utilities.randtest.`**`comp`**(*z1, z2, tol*)

Return a bool indicating whether the error between z1 and z2 is <= tol.

If z2 is non-zero and `|z1| > 1` the error is normalized by `|z1|`, so if you want the absolute error, call this as `comp(z1 - z2, 0, tol)`.

`sympy.utilities.randtest.`**`random_complex_number`**(*a=2, b=-1, c=3, d=1, rational=False*)

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee b <= Im z <= d, a <= Re z <= c

`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.**test_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 test_numerically as tn
>>> tn(sin(x)**2 + cos(x)**2, 1, x)
True
```

5.33.11 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*)

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 instance at 0x0000000003E64448>, 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 by looking harder for code in the case that it looks like the the code comes from a different module. In the case of decorated functions (e.g. @vectorize) they appear to come from a different module (e.g. multidemensional) even though their code is not there.

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 2041).

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 occuring 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 `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 `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.`**`isgeneratorfunction`**`(`*object*`)`

Return True if the object is a user-defined generator function.

Generator function objects provides same attributes as functions.

See isfunction.__doc__ for attributes listing.

Adapted from Python 2.6.

`sympy.utilities.runtests.`**`run_all_tests`**`(`*test_args=(), test_kwargs={}, doctest_args=(), doctest_kwargs={}, examples_args=(), examples_kwargs={'quiet': True}*`)`

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={}, command='C:\\Python27\\pytho module='sympy.utilities.runtest 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.**sympytestfile**(*filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=<doctest.DocTestParser instance at 0x0000000003E66448>, 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 `globs` 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" 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.

Explanation of test results

Output	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 verboseness can be set to "short" or "no" (default is "short")

```
>>> sympy.test(tb='no')
```

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 as well as the same architecture (32-bit vs. 64-bit).

5.33.12 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.

5.33.13 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.

5.34 Parsing input

5.34.1 Parsing Functions Reference

sympy.parsing.sympy_parser.**parse_expr**(*s*, *local_dict=None*, *transformations=(<function auto_symbol at 0x000000000E812CF8>, <function auto_number at 0x000000000E812E48>, <function factorial_notation at 0x000000000E812D68>)*, *global_dict=None*)

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!`).

See Also:

stringify_expr, eval_expr, standard_transformations, implicit_multiplication_application

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
```

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.sympy_tokenize.**printtoken**(*type*, *token*, *srow_scol*, *erow_ecol*, *line*)

sympy.parsing.sympy_tokenize.**tokenize**(*readline*, *tokeneater=<function printtoken at 0x000000000E812128>*)

The tokenize() function accepts two parameters: one representing the input stream, and one providing an output mechanism for tokenize().

The first parameter, readline, must be a callable object which provides the same interface as the readline() method of built-in file objects. Each call to the function should return one line of input as a string.

The second parameter, tokeneater, must also be a callable object. It is called once for each token, with five arguments, corresponding to the tuples generated by generate_tokens().

sympy.parsing.sympy_tokenize.**untokenize**(*iterable*)

Transform tokens back into Python source code.

Each element returned by the iterable must be a token sequence with at least two elements, a token number and token value. If only two tokens are passed, the resulting output is poor.

Round-trip invariant for full input: Untokenized source will match input source exactly

Round-trip invariant for limited intput:

```
# Output text will tokenize the back to the input
t1 = [tok[:2] for tok in generate_tokens(f.readline)]
newcode = untokenize(t1)
readline = iter(newcode.splitlines(1)).next
```

```
t2 = [tok[:2] for tok in generate_tokens(readline)]
assert t1 == t2
```

sympy.parsing.sympy_tokenize.**generate_tokens**(*readline*)

The generate_tokens() generator requires one argment, readline, which must be a callable object which provides the same interface as the readline() method of built-in file objects. Each call to the function should return one line of input as a string. Alternately, readline can be a callable function terminating with StopIteration:

```
readline = open(myfile).next    # Example of alternate readline
```

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed is the logical line; continuation lines are included.

sympy.parsing.sympy_tokenize.**group**(**choices*)

sympy.parsing.sympy_tokenize.**any**(**choices*)

sympy.parsing.sympy_tokenize.**maybe**(**choices*)

sympy.parsing.maxima.**parse_maxima**(*str*, *globals=None*, *name_dict={}*)

sympy.parsing.mathematica.**mathematica**(*s*)

5.34.2 Parsing Exceptions Reference

class sympy.parsing.sympy_tokenize.**TokenError**

class sympy.parsing.sympy_tokenize.**StopTokenizing**

5.34.3 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 auto_symbol at 0x00000000**

Standard transformations for `parse_expr()`. 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, _token_splittable,
... standard_transformations, implicit_multiplication,
... split_symbols_custom)
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...             return _token_splittable(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()`, otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Example:

```
>>> 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()`, otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Example:

```
>>> 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)
csc(z) + cot(z)
```

sympy.parsing.sympy_parser.**function_exponentiation**(*tokens, local_dict, global_dict*)

Allows functions to be exponentiated, e.g. `cos**2(x)`.

Example:

```
>>> 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.

Example:

```
>>> 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.**factorial_notation**(*tokens*, *local_dict*, *global_dict*)
Allows standard notation for factorial.

sympy.parsing.sympy_parser.**auto_symbol**(*tokens*, *local_dict*, *global_dict*)
Inserts calls to `Symbol` for undefined variables.

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`, float literals use `Float`, and repeating decimals use `Rational`.

5.35 Calculus

5.36 Physics Module

A module that helps solving problems in physics

5.36.1 Contents

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.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 1388)

References

[R207] (page 1629)

Examples

```
>>> from sympy.physics.gaussopt 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()
0.00100000000000000
>>> p.z_r.n()
5.92753330865999

>>> from sympy.physics.gaussopt 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.gaussopt import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.divergence
0.00053/pi
```

`gouy`

The Gouy phase.

Examples

```
>>> from sympy.physics.gaussopt 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.gaussopt 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.gaussopt import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.radius
0.2809/pi**2 + 1
```

w

The beam radius at $1/e^2$ intensity.

See Also:

`w_0` **(page 1384)** the minimal radius of beam

Examples

```
>>> from sympy.physics.gaussopt 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 1384)** the beam radius at $1/e^2$ intensity

Examples

```
>>> from sympy.physics.gaussopt import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w_0
0.00100000000000000
```

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.gaussopt import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.waist_approximation_limit
1.06e-6/pi
```

class sympy.physics.gaussopt.**CurvedMirror**

Ray Transfer Matrix for reflection from curved surface.

Parameters **R** : radius of curvature (positive for concave)

See Also:

RayTransferMatrix (page 1388)

Examples

```
>>> from sympy.physics.gaussopt import CurvedMirror
>>> from sympy import symbols
>>> R = symbols('R')
>>> CurvedMirror(R)
Matrix([
[   1, 0],
[-2/R, 1]])
```

class sympy.physics.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 1388)

Examples

```
>>> from sympy.physics.gaussopt 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]])
```

class sympy.physics.gaussopt.**FlatMirror**

Ray Transfer Matrix for reflection.

See Also:

RayTransferMatrix (page 1388)

Examples

```
>>> from sympy.physics.gaussopt import FlatMirror
>>> FlatMirror()
Matrix([
[1, 0],
[0, 1]])
```

class sympy.physics.gaussopt.**FlatRefraction**

Ray Transfer Matrix for refraction.

Parameters **n1** : refractive index of one medium

n2 : refractive index of other medium

See Also:

RayTransferMatrix (page 1388)

Examples

```
>>> from sympy.physics.gaussopt import FlatRefraction
>>> from sympy import symbols
>>> n1, n2 = symbols('n1 n2')
>>> FlatRefraction(n1, n2)
Matrix([
[1,     0],
[0, n1/n2]])
```

class sympy.physics.gaussopt.**FreeSpace**

Ray Transfer Matrix for free space.

Parameters **distance** :

See Also:

RayTransferMatrix (page 1388)

Examples

```
>>> from sympy.physics.gaussopt import FreeSpace
>>> from sympy import symbols
>>> d = symbols('d')
>>> FreeSpace(d)
Matrix([
[1, d],
[0, 1]])
```

class sympy.physics.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]))

Examples :

======= :

>>> from sympy.physics.gaussopt 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]]) :

See Also:

RayTransferMatrix (page 1388)

angle

The angle with the optical axis.

Examples

```
>>> from sympy.physics.gaussopt 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.gaussopt import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.height
h
```

class sympy.physics.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 1386), BeamParameter (page 1383), FreeSpace (page 1386), FlatRefraction (page 1386), CurvedRefraction (page 1385), FlatMirror (page 1386), CurvedMirror (page 1385), ThinLens (page 1389)

References

[R208] (page 1629)

Examples

```
>>> from sympy.physics.gaussopt 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
```

A

The A parameter of the Matrix.

Examples

```
>>> from sympy.physics.gaussopt import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.A
1
```

B

The B parameter of the Matrix.

Examples

```
>>> from sympy.physics.gaussopt import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.B
2
```

C

The C parameter of the Matrix.

Examples

```
>>> from sympy.physics.gaussopt import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.C
3
```

D

The D parameter of the Matrix.

Examples

```
>>> from sympy.physics.gaussopt import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.D
4
```

class sympy.physics.gaussopt.**ThinLens**

Ray Transfer Matrix for a thin lens.

Parameters **f** : the focal distance

See Also:

RayTransferMatrix (page 1388)

Examples

```
>>> from sympy.physics.gaussopt import ThinLens
>>> from sympy import symbols
>>> f = symbols('f')
>>> ThinLens(f)
Matrix([
[   1, 0],
[-1/f, 1]])
```

sympy.physics.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.gaussopt 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
```

sympy.physics.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.gaussopt 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.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 1391), geometric_conj_bf (page 1391)

Examples

```
>>> from sympy.physics.gaussopt import geometric_conj_ab
>>> from sympy import symbols
>>> a, b = symbols('a b')
>>> geometric_conj_ab(a, b)
a*b/(a + b)
```

sympy.physics.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 1391)

Examples

```
>>> from sympy.physics.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.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 1391)

Examples

```
>>> from sympy.physics.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.gaussopt.**rayleigh2waist**(*z_r, wavelen*)

Calculate the waist from the rayleigh range of a gaussian beam.

See Also:

waist2rayleigh (page 1392), BeamParameter (page 1383)

Examples

```
>>> from sympy.physics.gaussopt 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.gaussopt.**waist2rayleigh**(*w, wavelen*)

Calculate the rayleigh range from the waist of a gaussian beam.

See Also:

rayleigh2waist (page 1392), BeamParameter (page 1383)

Examples

```
>>> from sympy.physics.gaussopt import waist2rayleigh
>>> from sympy import symbols
>>> w, wavelen = symbols('w wavelen')
>>> waist2rayleigh(w, wavelen)
pi*w**2/wavelen
```

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.**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)
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.**mgamma**(*mu*, *lower=False*)

Returns a Dirac gamma matrix gamma^mu in the standard (Dirac) representation.

If you want gamma_mu, use gamma(mu, True).

We use a convention:

gamma^5 = I * gamma^0 * gamma^1 * gamma^2 * gamma^3 gamma_5 = I * gamma_0 * gamma_1 * gamma_2 * gamma_3 = - gamma^5

References

[R209] (page 1629)

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 sigma_i. i=1,2,3

References

[R210] (page 1629)

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

If the point we want the inertia about is a distance of 2 units of length and 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]])

In case we want to find the inertia along a vector of (1,1,1): >>> pat_matrix(2,1,1,1) Matrix([[4, -2, -2], [-2, 4, -2], [-2, -2, 4]])

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 hw, 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.**psi_n**(*n*, *x*, *m*, *omega*)

Returns the wavefunction psi_{n} for the One-dimensional harmonic oscillator.

n the "nodal" quantum number. Corresponds to the number of nodes in the wavefunction. n >= 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 >= 0

l the quantum number for orbital angular momentum

nu mass-scaled frequency: nu = m*omega/(2*hbar) where $m'isthemassand'omega$ 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, nu 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 follow 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 1399), sympy.functions.special.delta_functions.DiracDelta (page 336)

References

[R229] (page 1629)

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 1401), is_only_below_fermi (page 1401), is_only_above_fermi (page 1401)

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 1400), is_only_above_fermi (page 1401), is_only_below_fermi (page 1401)

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 1400), is_below_fermi (page 1401), is_only_below_fermi (page 1401)

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 1400), is_below_fermi (page 1401), is_only_above_fermi (page 1401)

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 1402)

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 1402)

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)
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
>>> 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 FockStateBosonBra (page 1407)

sympy.physics.secondquant.**BKet**
alias of FockStateBosonKet (page 1407)

sympy.physics.secondquant.**FBra**
alias of FockStateFermionBra

sympy.physics.secondquant.**FKet**
alias of FockStateFermionKet

sympy.physics.secondquant.**F**
alias of AnnihilateFermion (page 1403)

sympy.physics.secondquant.**Fd**
alias of CreateFermion (page 1405)

sympy.physics.secondquant.**B**
alias of AnnihilateBoson (page 1403)

sympy.physics.secondquant.**Bd**
alias of CreateBoson (page 1403)

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> and |b>, and leave the inner product unevaluated as <a|b>.

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
[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),F(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)
>>> NO(Fd(a)*Fd(i)).has_q_annihilators
-1
>>> NO(F(i)*F(a)).has_q_annihilators
1
>>> NO(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 NO, F, Fd

>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_creators
1
>>> NO(F(i)*F(a)).has_q_creators
-1
>>> NO(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 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]
```

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 inforation 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 permuation. 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] -> 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 1629).

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:

http://groups.google.com/group/sage-devel/browse_thread/thread/33835976efbb3b7f

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, prec=None*)

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 1629).

INPUT:

- `j_1`, `j_2`, `j_3`, `m_1`, `m_2`, `m_3` - 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 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:

$$\langle j_1 m_1 \; j_2 m_2 | j_3 m_3 \rangle = (-1)^{j_1 - j_2 + m_3} \sqrt{2j_3 + 1} \, Wigner3j(j_1, j_2, j_3, m_1, m_2, -m_3)$$

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.**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:

$$Y(j_1, j_2, j_3, m_1, m_2, m_3) = \int Y_{l_1,m_1}(\Omega) Y_{l_2,m_2}(\Omega) Y_{l_3,m_3}(\Omega) d\Omega = \sqrt{(2l_1+1)(2l_2+1)(2l_3+1)/(4\pi)} \, Y(j_1, j_2, j_3, 0, 0, 0) \; Y$$

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

$$Y(j_1, j_2, j_3, m_1, m_2, m_3) = Y(j_3, j_1, j_2, m_3, m_1, m_2) = Y(j_2, j_3, j_1, m_2, m_3, m_1) = Y(j_3, j_2, j_1, m_3, m_2, m_1) = Y(j_1, j$$

- invariant under space inflection, i.e.

$$Y(j_1, j_2, j_3, m_1, m_2, m_3) = Y(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 Regge symmetries as inherited for the $3j$ symbols [Regge58] (page 1629)
- 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. $J = l_1 + l_2 + l_3 = 2n$ for n in $\mathbb{N}$

ALGORITHM:

This function uses the algorithm of [Liberatodebrito82] (page 1629) 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 1629).

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:

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

Please see the 6j symbol for its much richer symmetries and for additional properties.

ALGORITHM:

This function uses the algorithm of [Edmonds74] (page 1629) 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 1629).

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, prec=None*)

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
- `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_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 $J := j_1 + j_2 + j_3$):

$$Wigner3j(j_1, j_2, j_3, m_1, m_2, m_3) = Wigner3j(j_3, j_1, j_2, m_3, m_1, m_2) = Wigner3j(j_2, j_3, j_1, m_2, m_3, m_1) = (-1)^J W$$

- invariant under space inflection, i.e.

$$Wigner3j(j_1, j_2, j_3, m_1, m_2, m_3) = (-1)^J Wigner3j(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 additional symmetries based on the work by [Regge58] (page 1629)

- 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 1629) 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 1629).

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 $Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6)$.

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.

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

The Wigner 6j symbol obeys the following symmetry rules:

- Wigner 6j symbols are left invariant under any permutation of the columns:

  $Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = Wigner6j(j_3, j_1, j_2, j_6, j_4, j_5) = Wigner6j(j_2, j_3, j_1, j_5, j_6, j_4) = Wigner6j(j_3, j_2,$

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

  $Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = Wigner6j(j_1, j_5, j_6, j_4, j_2, j_3) = Wigner6j(j_4, j_2, j_6, j_1, j_5, j_3) = Wigner6j(j_4, j_5,$

- additional 6 symmetries [Regge59] (page 1629) 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 1629) 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 1629).

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`, ..., `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...
```

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 1629) 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 1629).

Units

Introduction

This module provides around 200 predefined units that are commonly used in the sciences. Additionally, it provides the `Unit` class which allows you to define your own units.

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.physics.units import *
```

Dimensionless quantities Provides variables for commonly written dimensionless (unitless) quantities.

```
>>> 2*ten
20
>>> 20*percent
1/5
>>> 300*kilo*20*percent
60000
>>> nano*deg
pi/180000000000
```

Base units The SI base units are defined variable name that are commonly used in written and verbal communication. The singular abbreviated versions are what is used for display purposes, but the plural non-abbreviated versions are defined in case it helps readability.

```
>>> 5*meters
5*m
>>> milli*kilogram
kg/1000
>>> gram
kg/1000
```

Note that British Imperial and U.S. customary units are not included. We strongly urge the use of SI units; only Myanmar (Burma), Liberia, and the United States have not officially accepted the SI system.

Derived units Common SI derived units.

```
>>> joule
kg*m**2/s**2
```

Docstring

Physical units and dimensions.

The base class is Unit, where all here defined units (~200) inherit from.

The find_unit function can help you find units for a given quantity:

```
>>> import sympy.physics.units as u
>>> u.find_unit('coul')
['coulomb', 'coulombs']
>>> u.find_unit(u.charge)
['C', 'charge', 'coulomb', 'coulombs']
>>> u.coulomb
A*s
```

Units are always given in terms of base units that have a name and an abbreviation:

```
>>> u.A.name
'ampere'
>>> u.ampere.abbrev
'A'
```

The generic name for a unit (like 'length', 'mass', etc...) can help you find units:

```
>>> u.find_unit('magnet')
['magnetic_flux', 'magnetic_constant', 'magnetic_flux_density']
>>> u.find_unit(u.magnetic_flux)
['Wb', 'wb', 'weber', 'webers', 'magnetic_flux']
```

If, for a given session, you wish to add a unit you may do so:

```
>>> u.find_unit('gal')
[]
>>> u.gal = 4*u.quart
>>> u.gal/u.inch**3
231
```

To see a given quantity in terms of some other unit, divide by the desired unit:

```
>>> mph = u.miles/u.hours
>>> (u.m/u.s/mph).n(2)
2.2
```

The units are defined in terms of base units, so when you divide similar units you will obtain a pure number. This means, for example, that if you divide a real-world mass (like grams) by the atomic mass unit (amu) you will obtain Avogadro's number. To obtain the answer in moles you should divide by the unit `avogadro`:

```
>>> u.grams/u.amu
602214179000000000000000
>>> _/u.avogadro
mol
```

For chemical calculations the unit `mmu` (molar mass unit) has been defined so this conversion is handled automatically. For example, the number of moles in 1 kg of water might be calculated as:

```
>>> u.kg/(18*u.mmu).n(3)
55.5*mol
```

If you need the number of atoms in a mol as a pure number you can use `avogadro_number` but if you need it as a dimensional quantity you should use `avogadro_constant`. (`avogadro` is a shorthand for the dimensional quantity.)

```
>>> u.avogadro_number
602214179000000000000000
>>> u.avogadro_constant
602214179000000000000000/mol
```

class sympy.physics.units.**Unit**

Base class for base unit of physical units.

```
>>> from sympy.physics.units import Unit
>>> Unit("meter", "m")
m
```

Other units are derived from base units:

```
>>> import sympy.physics.units as u
>>> cm = u.m/100
>>> 100*u.cm
m
```

sympy.physics.units.**find_unit**(*quantity*)

Return a list of matching units names. if quantity is a string – units containing the string *quantity* if quantity is a unit – units having matching base units

Examples

```
>>> from sympy.physics import units as u
>>> u.find_unit('charge')
['charge']
>>> u.find_unit(u.charge)
['C', 'charge', 'coulomb', 'coulombs']
>>> u.find_unit('volt')
['volt', 'volts', 'voltage']
>>> u.find_unit(u.inch**3)[:5]
['l', 'cl', 'dl', 'ml', 'liter']
```

High energy physics

Abstract

Contains docstrings for methods in high energy physics.

Gamma matrices

High energy physics

Classical Mechanics

Authors Gilbert Gede, Luke Peterson, Angadh Nanjangud

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.

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

Mechanics: Vector & ReferenceFrame In `mechanics`, 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 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:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$$
$$(\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c})$$

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 `mechanics`: 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 **a** and **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 **c** has the following properties: it's orientation is perpendicular to both **a** and **b**, it's magnitude is defined as $\|\mathbf{c}\| = \|\mathbf{a}\|\|\mathbf{b}\| \sin(\theta)$ (where θ is the angle between **a** and **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:

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$$
$$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$$

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 not be discussed here, but instead in the inertia section (that is where it is used). Other useful vector properties and relationships are:

$$\alpha(\mathbf{a} + \mathbf{b}) = \alpha\mathbf{a} + \alpha\mathbf{b}$$
$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$$
$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{b}$$

$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$ gives the scalar triple product.

$\mathbf{a} \times (\mathbf{b} \cdot \mathbf{c})$ does not work, as you cannot cross a vector and a scalar.

$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$$
$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = (\mathbf{b} \times \mathbf{c}) \cdot \mathbf{a} = (\mathbf{c} \times \mathbf{a}) \cdot \mathbf{b}$$
$$(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{a}(\mathbf{b} \cdot \mathbf{c})$$
$$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b})$$

Alternative Representation If we have three non-coplanar unit vectors $\hat{\mathbf{n}}_\mathbf{x}, \hat{\mathbf{n}}_\mathbf{y}, \hat{\mathbf{n}}_\mathbf{z}$, we can represent any vector $\mathbf{a}$ as $\mathbf{a} = a_x\hat{\mathbf{n}}_\mathbf{x} + a_y\hat{\mathbf{n}}_\mathbf{y} + a_z\hat{\mathbf{n}}_\mathbf{z}$. In this situation $\hat{\mathbf{n}}_\mathbf{x}, \hat{\mathbf{n}}_\mathbf{y}, \hat{\mathbf{n}}_\mathbf{z}$ 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:

$$\mathbf{a} = a_x\hat{\mathbf{n}}_\mathbf{x} + a_y\hat{\mathbf{n}}_\mathbf{y} + a_z\hat{\mathbf{n}}_\mathbf{z}$$
$$\mathbf{b} = b_x\hat{\mathbf{n}}_\mathbf{x} + b_y\hat{\mathbf{n}}_\mathbf{y} + b_z\hat{\mathbf{n}}_\mathbf{z}$$

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:

$$\mathbf{a} + \mathbf{b} = (a_x + b_x)\hat{\mathbf{n}}_{\mathbf{x}} + (a_y + b_y)\hat{\mathbf{n}}_{\mathbf{y}} + (a_z + b_z)\hat{\mathbf{n}}_{\mathbf{z}}$$

Multiplication operations are now defined as:

$$\begin{aligned}
\alpha\mathbf{b} &= \alpha b_x\hat{\mathbf{n}}_{\mathbf{x}} + \alpha b_y\hat{\mathbf{n}}_{\mathbf{y}} + \alpha b_z\hat{\mathbf{n}}_{\mathbf{z}} \\
\mathbf{a}\cdot\mathbf{b} &= a_x b_x + a_y b_y + a_z b_z \\
\mathbf{a}\times\mathbf{b} &= \det\begin{bmatrix} \hat{\mathbf{n}}_{\mathbf{x}} & \hat{\mathbf{n}}_{\mathbf{y}} & \hat{\mathbf{n}}_{\mathbf{z}} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} \\
(\mathbf{a}\times\mathbf{b})\cdot\mathbf{c} &= \det\begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}
\end{aligned}$$

To write a vector in a given basis, we can do the follow:

$$\mathbf{a} = (\mathbf{a}\cdot\hat{\mathbf{n}}_{\mathbf{x}})\hat{\mathbf{n}}_{\mathbf{x}} + (\mathbf{a}\cdot\hat{\mathbf{n}}_{\mathbf{y}})\hat{\mathbf{n}}_{\mathbf{y}} + (\mathbf{a}\cdot\hat{\mathbf{n}}_{\mathbf{z}})\hat{\mathbf{n}}_{\mathbf{z}}$$

Examples Some numeric examples of these operations follow:

$$\begin{aligned}
\mathbf{a} &= \hat{\mathbf{n}}_{\mathbf{x}} + 5\hat{\mathbf{n}}_{\mathbf{y}} \\
\mathbf{b} &= \hat{\mathbf{n}}_{\mathbf{y}} + \alpha\hat{\mathbf{n}}_{\mathbf{z}} \\
\mathbf{a}+\mathbf{b} &= \hat{\mathbf{n}}_{\mathbf{x}} + 6\hat{\mathbf{n}}_{\mathbf{y}} + \alpha\hat{\mathbf{n}}_{\mathbf{z}} \\
\mathbf{a}\cdot\mathbf{b} &= 5 \\
\mathbf{a}\cdot\hat{\mathbf{n}}_{\mathbf{y}} &= 5 \\
\mathbf{a}\cdot\hat{\mathbf{n}}_{\mathbf{z}} &= 0 \\
\mathbf{a}\times\mathbf{b} &= 5\alpha\hat{\mathbf{n}}_{\mathbf{x}} - \alpha\hat{\mathbf{n}}_{\mathbf{y}} + \hat{\mathbf{n}}_{\mathbf{z}} \\
\mathbf{b}\times\mathbf{a} &= -5\alpha\hat{\mathbf{n}}_{\mathbf{x}} + \alpha\hat{\mathbf{n}}_{\mathbf{y}} - \hat{\mathbf{n}}_{\mathbf{z}}
\end{aligned}$$

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 **N**, vector **a** is said to be fixed in the frame **N** if none of its properties ever change when observed from **N**. We will typically assign a fixed orthonormal basis vector set with each reference frame; **N** will have $\hat{\mathbf{n}}_{\mathbf{x}}, \hat{\mathbf{n}}_{\mathbf{y}}, \hat{\mathbf{n}}_{\mathbf{z}}$ 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.

Fixed in:

	A	B
c	x	
d	x	
e		x
f		x

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:

$$\frac{{}^{\mathbf{A}}d}{dt}(\mathbf{a}+\mathbf{b}) = \frac{{}^{\mathbf{A}}d\mathbf{a}}{dt} + \frac{{}^{\mathbf{A}}d\mathbf{b}}{dt}$$
$$\frac{{}^{\mathbf{A}}d}{dt}\gamma\mathbf{a} = \frac{d\gamma}{dt}\mathbf{a} + \gamma\frac{{}^{\mathbf{A}}d\mathbf{a}}{dt}$$
$$\frac{{}^{\mathbf{A}}d}{dt}(\mathbf{a}\times\mathbf{b}) = \frac{{}^{\mathbf{A}}d\mathbf{a}}{dt}\times\mathbf{b} + \mathbf{a}\times\frac{{}^{\mathbf{A}}d\mathbf{b}}{dt}$$

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:

$$\begin{bmatrix}\hat{\mathbf{a}}_{\mathbf{x}}\\ \hat{\mathbf{a}}_{\mathbf{y}}\\ \hat{\mathbf{a}}_{\mathbf{z}}\end{bmatrix} = \left[{}^{\mathbf{A}}\mathbf{C}^{\mathbf{B}}\right]\begin{bmatrix}\hat{\mathbf{b}}_{\mathbf{x}}\\ \hat{\mathbf{b}}_{\mathbf{y}}\\ \hat{\mathbf{b}}_{\mathbf{z}}\end{bmatrix}$$

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 $\hat{\mathbf{a}}_{\mathbf{z}}$ and $\hat{\mathbf{b}}_{\mathbf{z}}$ 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:

$$\begin{bmatrix}C_{xx} & C_{xy} & C_{xz}\\ C_{yx} & C_{yy} & C_{yz}\\ C_{zx} & C_{zy} & C_{zz}\end{bmatrix} = \begin{bmatrix}\hat{\mathbf{a}}_{\mathbf{x}}\cdot\hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{x}}\cdot\hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{x}}\cdot\hat{\mathbf{b}}_{\mathbf{z}}\\ \hat{\mathbf{a}}_{\mathbf{y}}\cdot\hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{y}}\cdot\hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{y}}\cdot\hat{\mathbf{b}}_{\mathbf{z}}\\ \hat{\mathbf{a}}_{\mathbf{z}}\cdot\hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{z}}\cdot\hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{z}}\cdot\hat{\mathbf{b}}_{\mathbf{z}}\end{bmatrix}$$

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: $\mathbf{a} = \hat{\mathbf{a}}_\mathbf{x} + \hat{\mathbf{a}}_\mathbf{y} + \hat{\mathbf{a}}_\mathbf{z}$ and $\mathbf{b} = \hat{\mathbf{b}}_\mathbf{x} + \hat{\mathbf{b}}_\mathbf{y} + \hat{\mathbf{b}}_\mathbf{z}$. If we wish to express **b** in the **A** frame, we do the following:

$$\begin{aligned}
\mathbf{b} &= \hat{\mathbf{b}}_\mathbf{x} + \hat{\mathbf{b}}_\mathbf{y} + \hat{\mathbf{b}}_\mathbf{z} \\
\mathbf{b} &= \left[\hat{\mathbf{a}}_\mathbf{x} \cdot (\hat{\mathbf{b}}_\mathbf{x} + \hat{\mathbf{b}}_\mathbf{y} + \hat{\mathbf{b}}_\mathbf{z})\right] \hat{\mathbf{a}}_\mathbf{x} + \left[\hat{\mathbf{a}}_\mathbf{y} \cdot (\hat{\mathbf{b}}_\mathbf{x} + \hat{\mathbf{b}}_\mathbf{y} + \hat{\mathbf{b}}_\mathbf{z})\right] \hat{\mathbf{a}}_\mathbf{y} + \left[\hat{\mathbf{a}}_\mathbf{z} \cdot (\hat{\mathbf{b}}_\mathbf{x} + \hat{\mathbf{b}}_\mathbf{y} + \hat{\mathbf{b}}_\mathbf{z})\right] \hat{\mathbf{a}}_\mathbf{z} \\
\mathbf{b} &= (\cos(\theta) - \sin(\theta))\hat{\mathbf{a}}_\mathbf{x} + (\sin(\theta) + \cos(\theta))\hat{\mathbf{a}}_\mathbf{y} + \hat{\mathbf{a}}_\mathbf{z}
\end{aligned}$$

And if we wish to express **a** in the **B**, we do:

$$\begin{aligned}
\mathbf{a} &= \hat{\mathbf{a}}_\mathbf{x} + \hat{\mathbf{a}}_\mathbf{y} + \hat{\mathbf{a}}_\mathbf{z} \\
\mathbf{a} &= \left[\hat{\mathbf{b}}_\mathbf{x} \cdot (\hat{\mathbf{a}}_\mathbf{x} + \hat{\mathbf{a}}_\mathbf{y} + \hat{\mathbf{a}}_\mathbf{z})\right] \hat{\mathbf{b}}_\mathbf{x} + \left[\hat{\mathbf{b}}_\mathbf{y} \cdot (\hat{\mathbf{a}}_\mathbf{x} + \hat{\mathbf{a}}_\mathbf{y} + \hat{\mathbf{a}}_\mathbf{z})\right] \hat{\mathbf{b}}_\mathbf{y} + \left[\hat{\mathbf{b}}_\mathbf{z} \cdot (\hat{\mathbf{a}}_\mathbf{x} + \hat{\mathbf{a}}_\mathbf{y} + \hat{\mathbf{a}}_\mathbf{z})\right] \hat{\mathbf{b}}_\mathbf{z} \\
\mathbf{a} &= (\cos(\theta) + \sin(\theta))\hat{\mathbf{b}}_\mathbf{x} + (-\sin(\theta) + \cos(\theta))\hat{\mathbf{b}}_\mathbf{y} + \hat{\mathbf{b}}_\mathbf{z}
\end{aligned}$$

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: $\mathbf{a} = a_x\hat{\mathbf{a}}_\mathbf{x} + a_y\hat{\mathbf{a}}_\mathbf{y} + a_z\hat{\mathbf{a}}_\mathbf{z}$ and $\mathbf{b} = b_x\hat{\mathbf{b}}_\mathbf{x} + b_y\hat{\mathbf{b}}_\mathbf{y} + b_z\hat{\mathbf{b}}_\mathbf{z}$. If we want to take the derivative of **b** in the reference frame **A**, we must first express it in **A**, and the take the derivatives of the measure numbers:

$$\frac{{}^{\mathbf{A}}d\mathbf{b}}{dx} = \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_\mathbf{x})}{dx}\hat{\mathbf{a}}_\mathbf{x} + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_\mathbf{y})}{dx}\hat{\mathbf{a}}_\mathbf{y} + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_\mathbf{z})}{dx}\hat{\mathbf{a}}_\mathbf{z} +$$

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 $\mathbf{c}$; $\mathbf{c} = x\hat{\mathbf{b}}_{\mathbf{x}} + l\hat{\mathbf{b}}_{\mathbf{y}}$. This provides a definition in the $\mathbf{B}$ frame. We can now do the following:

$$\begin{aligned}\frac{{}^{\mathbf{B}}d\mathbf{c}}{dt} &= \frac{dx}{dt}\hat{\mathbf{b}}_{\mathbf{x}} + \frac{dl}{dt}\hat{\mathbf{b}}_{\mathbf{y}} \\ &= \dot{x}\hat{\mathbf{b}}_{\mathbf{x}}\end{aligned}$$

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:

$$\begin{aligned}\frac{{}^{\mathbf{A}}d\mathbf{c}}{dt} &= \frac{d(\mathbf{c}\cdot\hat{\mathbf{a}}_{\mathbf{x}})}{dt}\hat{\mathbf{a}}_{\mathbf{x}} + \frac{d(\mathbf{c}\cdot\hat{\mathbf{a}}_{\mathbf{y}})}{dt}\hat{\mathbf{a}}_{\mathbf{y}} + \frac{d(\mathbf{c}\cdot\hat{\mathbf{a}}_{\mathbf{z}})}{dt}\hat{\mathbf{a}}_{\mathbf{z}} \\ &= \frac{d(\cos(\theta)x)}{dt}\hat{\mathbf{a}}_{\mathbf{x}} + \frac{d(l)}{dt}\hat{\mathbf{a}}_{\mathbf{y}} + \frac{d(-\sin(\theta)x)}{dt}\hat{\mathbf{a}}_{\mathbf{z}} \\ &= (-\dot{\theta}\sin(\theta)x + \cos(\theta)\dot{x})\hat{\mathbf{a}}_{\mathbf{x}} + (\dot{\theta}\cos(\theta)x + \sin(\theta)\dot{x})\hat{\mathbf{a}}_{\mathbf{z}}\end{aligned}$$

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 in SymPy's Mechanics 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 `mechanics`, one of the first steps is defining a reference frame (remember to import sympy.physics.mechanics first):

```
>>> from sympy.physics.mechanics 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 and 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 Mechanics 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
x*N.x
>>> x*(N.x + N.y)
x*N.x + x*N.y
```

In `mechanics` 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 `mechanics` for vector multiplication do not posses 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)
```

Vector Calculus, in Mechanics 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 `mechanics` 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
A.x + N.x
```

If we want to do vector multiplication, first we have to define and 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 ${}^{\mathbf{N}}\mathbf{C}^{\mathbf{A}}$.

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
[sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), -sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), -sin(q1)*cos
[sin(q1)*sin(q3) - sin(q2)*cos(q1)*cos(q3),  sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1),  cos(q1)*cos
>>> B.orient(N,'Body',[q1,q2,q3],'XYZ')
>>> N.dcm(B)
Matrix([
[                          cos(q2)*cos(q3),                           -sin(q3)*cos(q2),          sin
[sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), -sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), -sin(q1)*cos
[sin(q1)*sin(q3) - sin(q2)*cos(q1)*cos(q3),  sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1),  cos(q1)*cos
```

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 1629).

Finally, before starting multiframe calculus operations, we will introduce another `mechanics` 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 `mprint` instead of print for non-interactive sessions.

```
>>> q1
q1(t)
>>> q1d = diff(q1, Symbol('t'))
>>> mprint(q1)
q1
>>> mprint(q1d)
q1'
```

For interactive sessions use `mechanics_printing`. There also exist analogs for SymPy's `pprint`, `mpprint`, and `latex`, `mlatex`.

```
>>> mechanics_printing()
>>> q1
q1
>>> q1d
q1'
```

A 'dynamicsymbol' should be used to represent any time varying quantity in `mechanics`, 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)
B.y + N.x
```

How Vectors are Coded in Mechanics What follows is a short description of how vectors are defined by the code in `mechanics`. It is provided for those who want to learn more about how this part of `mechanics` 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 also you also are setting `N`'s orientation dictionary to include `A` and its `Matrix` (that DCM being the transpose of the other).

Mechanics: Kinematics In mechanics, kinematics refers to the motion of bodies (or particles). It is the 'a' in 'f=ma'. This document will give some mathematical background to describing a system's kinematics as well as how to represent the kinematics in `mechanics`.

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 **B** in **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:

$$ {}^{\mathbf{N}}\omega^{\mathbf{B}} = (\frac{{}^{\mathbf{N}}d\hat{\mathbf{b}}_{\mathbf{y}}}{dt} \cdot \hat{\mathbf{b}}_{\mathbf{z}})\hat{\mathbf{b}}_{\mathbf{x}} + (\frac{{}^{\mathbf{N}}d\hat{\mathbf{b}}_{\mathbf{z}}}{dt} \cdot \hat{\mathbf{b}}_{\mathbf{x}})\hat{\mathbf{b}}_{\mathbf{y}} + (\frac{{}^{\mathbf{N}}d\hat{\mathbf{b}}_{\mathbf{x}}}{dt} \cdot \hat{\mathbf{b}}_{\mathbf{y}})\hat{\mathbf{b}}_{\mathbf{z}} $$

It is also common for a body's angular velocity to be written as:

$$^{\mathbf{N}}\omega^{\mathbf{B}} = w_x\hat{\mathbf{b}}_{\mathbf{x}} + w_y\hat{\mathbf{b}}_{\mathbf{y}} + w_z\hat{\mathbf{b}}_{\mathbf{z}}$$

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:

$$^{\mathbf{N}}\omega^{\mathbf{D}} = {}^{\mathbf{N}}\omega^{\mathbf{A}} + {}^{\mathbf{A}}\omega^{\mathbf{B}} + {}^{\mathbf{B}}\omega^{\mathbf{C}} + {}^{\mathbf{C}}\omega^{\mathbf{D}}$$

This is also shown in the following example:

$$^{\mathbf{N}}\omega^{\mathbf{A}} = 0$$
$$^{\mathbf{A}}\omega^{\mathbf{B}} = \dot{q}_1\hat{\mathbf{a}}_{\mathbf{x}}$$
$$^{\mathbf{B}}\omega^{\mathbf{C}} = -\dot{q}_2\hat{\mathbf{b}}_{\mathbf{z}}$$
$$^{\mathbf{C}}\omega^{\mathbf{D}} = \dot{q}_3\hat{\mathbf{c}}_{\mathbf{y}}$$
$$^{\mathbf{N}}\omega^{\mathbf{D}} = \dot{q}_1\hat{\mathbf{a}}_{\mathbf{x}} - \dot{q}_2\hat{\mathbf{b}}_{\mathbf{z}} + \dot{q}_3\hat{\mathbf{c}}_{\mathbf{y}}$$

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**:

$$^{\mathbf{A}}\omega^{\mathbf{D}} = w_1\hat{\mathbf{p_1}} + w_2\hat{\mathbf{p_2}} + w_3\hat{\mathbf{p_3}}$$

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 **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 **N**, we take it in **B**, where **B** can be any reference frame or body, usually one in which it is easy to take the derivative on **v** in (**v** is usually composed only of the basis vector set belonging to **B**). Then we add the cross product of the angular velocity of our newer frame, $^{\mathbf{N}}\omega^{\mathbf{B}}$ and our vector quantity **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 **B** in **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:

$$
\begin{aligned}
{}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \frac{{}^{\mathbf{N}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} \\
{}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}} \\
\text{if } {}^{\mathbf{N}}\omega^{\mathbf{B}} &= w_x\hat{\mathbf{b}}_{\mathbf{x}} + w_y\hat{\mathbf{b}}_{\mathbf{y}} + w_z\hat{\mathbf{b}}_{\mathbf{z}} \\
\text{then } {}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \frac{{}^{\mathbf{B}}d{}^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + \underbrace{{}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}}_{\text{this is 0 by definition}} \\
{}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \frac{dw_x}{dt}\hat{\mathbf{b}}_{\mathbf{x}} + \frac{dw_y}{dt}\hat{\mathbf{b}}_{\mathbf{y}} + \frac{dw_z}{dt}\hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \dot{w}_x\hat{\mathbf{b}}_{\mathbf{x}} + \dot{w}_y\hat{\mathbf{b}}_{\mathbf{y}} + \dot{w}_z\hat{\mathbf{b}}_{\mathbf{z}}
\end{aligned}
$$

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:

$$
\begin{aligned}
\mathbf{r}^{OP}&\text{, the position vector from } O \text{ to } P \\
{}^{\mathbf{N}}\mathbf{v}^{P}&\text{, the velocity of } P \text{ in the reference frame } \mathbf{N} \\
{}^{\mathbf{N}}\mathbf{a}^{P}&\text{, the acceleration of } P \text{ in the reference frame } \mathbf{N}
\end{aligned}
$$

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:

$$\text{Let's define: } \mathbf{r}^{OP} = q_x\hat{\mathbf{n}}_\mathbf{x} + q_y\hat{\mathbf{n}}_\mathbf{y}$$
$$^{\mathbf{N}}\mathbf{v}^P = \frac{^{\mathbf{N}}d\mathbf{r}^{OP}}{dt}$$
$$\text{then we can calculate: } ^{\mathbf{N}}\mathbf{v}^P = \dot{q}_x\hat{\mathbf{n}}_\mathbf{x} + \dot{q}_y\hat{\mathbf{n}}_\mathbf{y}$$
$$\text{and :} ^{\mathbf{N}}\mathbf{a}^P = \frac{^{\mathbf{N}}d^{\mathbf{N}}\mathbf{v}^P}{dt}$$
$$^{\mathbf{N}}\mathbf{a}^P = \ddot{q}_x\hat{\mathbf{n}}_\mathbf{x} + \ddot{q}_y\hat{\mathbf{n}}_\mathbf{y}$$

It is critical to understand in the above example that the point O is fixed in the reference frame **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:

$$\text{Let us instead define the velocity vector as: } ^{\mathbf{N}}\mathbf{v}^P = u_x\hat{\mathbf{n}}_\mathbf{x} + u_y\hat{\mathbf{n}}_\mathbf{y}$$
$$\text{then acceleration can be written as: } ^{\mathbf{N}}\mathbf{a}^P = \dot{u}_x\hat{\mathbf{n}}_\mathbf{x} + \dot{u}_y\hat{\mathbf{n}}_\mathbf{y}$$

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:

$$
\begin{aligned}
{}^{\mathbf{N}}\mathbf{v}^P &= {}^{\mathbf{N}}\mathbf{v}^S + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP} \\
{}^{\mathbf{N}}\mathbf{a}^P &= {}^{\mathbf{N}}\mathbf{a}^S + {}^{\mathbf{N}}\alpha^{\mathbf{B}} \times \mathbf{r}^{SP} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times ({}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP})
\end{aligned}
$$

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:

$$
\begin{aligned}
{}^{\mathbf{N}}\mathbf{v}^P &= {}^{\mathbf{B}}\mathbf{v}^P + {}^{\mathbf{N}}\mathbf{v}^S + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP} \\
{}^{\mathbf{N}}\mathbf{a}^P &= {}^{\mathbf{B}}\mathbf{a}^S + {}^{\mathbf{N}}\mathbf{a}^O + {}^{\mathbf{N}}\alpha^{\mathbf{B}} \times \mathbf{r}^{SP} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times ({}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP}) + 2{}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{B}}\mathbf{v}^P
\end{aligned}
$$

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:

$$
\begin{aligned}
{}^{\mathbf{N}}\omega^{\mathbf{B}} &= u_3\hat{\mathbf{n}}_{\mathbf{z}} = u_3\hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{v}^{O} &= u_1\hat{\mathbf{n}}_{\mathbf{x}} + u_2\hat{\mathbf{n}}_{\mathbf{y}}
\end{aligned}
$$

and accelerations can be written as:

$$
\begin{aligned}
{}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \dot{u}_3\hat{\mathbf{n}}_{\mathbf{z}} = \dot{u}_3\hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{a}^{O} &= \dot{u}_1\hat{\mathbf{n}}_{\mathbf{x}} + \dot{u}_2\hat{\mathbf{n}}_{\mathbf{y}}
\end{aligned}
$$

We can use the 2 point theorem to calculate the velocity and acceleration of point P now.

$$
\begin{aligned}
&\mathbf{r}^{OP} = R\hat{\mathbf{b}}_{\mathbf{x}} \\
&{}^{\mathbf{N}}\mathbf{v}^{P} = {}^{\mathbf{N}}\mathbf{v}^{O} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{OP} \\
&{}^{\mathbf{N}}\mathbf{v}^{P} = u_1\hat{\mathbf{n}}_{\mathbf{x}} + u_2\hat{\mathbf{n}}_{\mathbf{y}} + u_3\hat{\mathbf{b}}_{\mathbf{z}} \times R\hat{\mathbf{b}}_{\mathbf{x}} = u_1\hat{\mathbf{n}}_{\mathbf{x}} + u_2\hat{\mathbf{n}}_{\mathbf{y}} + u_3R\hat{\mathbf{b}}_{\mathbf{y}} \\
&{}^{\mathbf{N}}\mathbf{a}^{P} = {}^{\mathbf{N}}\mathbf{a}^{O} + {}^{\mathbf{N}}\alpha^{\mathbf{B}} \times \mathbf{r}^{OP} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times ({}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{OP}) \\
&{}^{\mathbf{N}}\mathbf{a}^{P} = \dot{u}_1\hat{\mathbf{n}}_{\mathbf{x}} + \dot{u}_2\hat{\mathbf{n}}_{\mathbf{y}} + \dot{u}_3\hat{\mathbf{b}}_{\mathbf{z}} \times R\hat{\mathbf{b}}_{\mathbf{x}} + u_3\hat{\mathbf{b}}_{\mathbf{z}} \times (u_3\hat{\mathbf{b}}_{\mathbf{z}} \times R\hat{\mathbf{b}}_{\mathbf{x}}) \\
&{}^{\mathbf{N}}\mathbf{a}^{P} = \dot{u}_1\hat{\mathbf{n}}_{\mathbf{x}} + \dot{u}_2\hat{\mathbf{n}}_{\mathbf{y}} + R\dot{u}_3\hat{\mathbf{b}}_{\mathbf{y}} - Ru_3^2\hat{\mathbf{b}}_{\mathbf{x}}
\end{aligned}
$$

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 $\mathbf{N}$.

$$
\begin{aligned}
\mathbf{r}^{OQ} &= l\hat{\mathbf{b}}_{\mathbf{x}} \\
\mathbf{r}^{QP} &= l\hat{\mathbf{c}}_{\mathbf{x}} \\
{}^{\mathbf{N}}\omega^{\mathbf{B}} &= u_1\hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\omega^{\mathbf{C}} &= u_2\hat{\mathbf{c}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{v}^{Q} &= {}^{\mathbf{N}}\mathbf{v}^{O} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{OQ} \\
{}^{\mathbf{N}}\mathbf{v}^{Q} &= u_1 l\hat{\mathbf{b}}_{\mathbf{y}} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= {}^{\mathbf{N}}\mathbf{v}^{Q} + {}^{\mathbf{N}}\omega^{\mathbf{C}} \times \mathbf{r}^{QP} \\
{}^{\mathbf{N}}\mathbf{v}^{Q} &= u_1 l\hat{\mathbf{b}}_{\mathbf{y}} + u_2\hat{\mathbf{c}}_{\mathbf{z}} \times l\hat{\mathbf{c}}_{\mathbf{x}} \\
{}^{\mathbf{N}}\mathbf{v}^{Q} &= u_1 l\hat{\mathbf{b}}_{\mathbf{y}} + u_2 l\hat{\mathbf{c}}_{\mathbf{y}}
\end{aligned}
$$

In this example we have a particle moving on a ring; the ring is supported by a rod which can rotate about the $\hat{\mathbf{n}}_{\mathbf{x}}$ 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.

$$
\begin{aligned}
{}^{\mathbf{N}}\omega^{\mathbf{C}} &= u_1\hat{\mathbf{n}}_{\mathbf{x}} \\
\mathbf{r}^{OQ} &= -l\hat{\mathbf{c}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{v}^{Q} &= u_1 l\hat{\mathbf{c}}_{\mathbf{y}} \\
\mathbf{r}^{QP} &= R(cos(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + sin(q_2)\hat{\mathbf{c}}_{\mathbf{y}}) \\
{}^{\mathbf{C}}\mathbf{v}^{P} &= Ru_2(-sin(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + cos(q_2)\hat{\mathbf{c}}_{\mathbf{y}}) \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= {}^{\mathbf{C}}\mathbf{v}^{P} + {}^{\mathbf{N}}\mathbf{v}^{Q} + {}^{\mathbf{N}}\omega^{\mathbf{C}} \times \mathbf{r}^{QP} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= Ru_2(-sin(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + cos(q_2)\hat{\mathbf{c}}_{\mathbf{y}}) + u_1 l\hat{\mathbf{c}}_{\mathbf{y}} + u_1\hat{\mathbf{c}}_{\mathbf{x}} \times R(cos(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + sin(q_2)\hat{\mathbf{c}}_{\mathbf{y}} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= -Ru_2 sin(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + (Ru_2 cos(q_2) + u_1 l)\hat{\mathbf{c}}_{\mathbf{y}} + Ru_1 sin(q_2)\hat{\mathbf{c}}_{\mathbf{z}}
\end{aligned}
$$

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 SymPy's Mechanics 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 `mechanics` 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 612).

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.mechanics import *
>>> mechanics_printing()
>>> 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)
u3*C.y - u2*B.z + u1*A.x
```

In `mechanics` 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)
l*u1*C.y
>>> P.v1pt_theory(Q, N, C)
- R*u2*sin(q2)*C.x + (R*u2*cos(q2) + l*u1)*C.y + R*u1*sin(q2)*C.z
```

Mechanics: Masses, Inertias, Particles and Rigid Bodies 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 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
```

Notice that the `inertia` function returns a dyadic with each component represented as two unit vectors separated by a `|`. Refer to the *Dyadic* (page 1453) section for more information about dyadics.

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')
>>> 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 1629), [WikiDyadics] (page 1629), [WikiDyadicProducts] (page 1629)). 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:

$$\hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{a}}_{\mathbf{x}} = \hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{x}}$$
$$\hat{\mathbf{a}}_{\mathbf{x}} \otimes \hat{\mathbf{a}}_{\mathbf{y}} = \hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{y}}$$

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:

$$(x\mathbf{v}) \otimes \mathbf{w} = \mathbf{v} \otimes (x\mathbf{w}) = x(\mathbf{v} \otimes \mathbf{w})$$
$$\mathbf{v} \otimes (\mathbf{w} + \mathbf{u}) = \mathbf{v} \otimes \mathbf{w} + \mathbf{v} \otimes \mathbf{u}$$
$$(\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} = \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u}$$

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:

$$\begin{aligned}
\hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{a}}_{\mathbf{x}} &= \hat{\mathbf{a}}_{\mathbf{x}} \\
\hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{a}}_{\mathbf{x}} &= \hat{\mathbf{a}}_{\mathbf{y}} \\
\hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{a}}_{\mathbf{x}} &= 0 \\
\hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{x}} &= \hat{\mathbf{a}}_{\mathbf{x}} \\
\hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{y}} &= \hat{\mathbf{a}}_{\mathbf{y}} \\
\hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{x}} &= 0 \\
\hat{\mathbf{a}}_{\mathbf{x}} \times \hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{x}} &= \hat{\mathbf{a}}_{\mathbf{z}}\hat{\mathbf{a}}_{\mathbf{x}} \\
\hat{\mathbf{a}}_{\mathbf{x}} \times \hat{\mathbf{a}}_{\mathbf{x}}\hat{\mathbf{a}}_{\mathbf{x}} &= 0 \\
\hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{x}} \times \hat{\mathbf{a}}_{\mathbf{z}} &= -\hat{\mathbf{a}}_{\mathbf{y}}\hat{\mathbf{a}}_{\mathbf{y}}
\end{aligned}$$

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 symp.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 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 symp.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*q1d**2
>> kinetic_energy(N, Pa, A)
M*l1**2*q1d**2/2 + 2*l1**2*m*q1d**2 + q1d**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 `set_potential_energy` method. The potential energy of any number of entities comprising the system can then be determined:

```
>> Pa.set_potential_energy(m * g * h)
>> A.set_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*q1d**2/2 - g*h*m + 2*l1**2*m*q1d**2 + q1d**2/2
```

Please refer to the docstrings to learn more about each function.

Kane's Method in Physics/Mechanics `mechanics` has been written for use with Kane's method of forming equations of motion [Kane1985] (page 1629). 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:

$$\mathbf{M}(q,t) = \begin{bmatrix} \mathbf{k_d}(q,t) \\ \mathbf{k_{dnh}}(q,t) \end{bmatrix}$$
$$_{(\mathbf{forcing})}(q,\dot{q},u,t) = \begin{bmatrix} -\mathbf{f_d}(q,\dot{q},u,t) \\ -\mathbf{f_{dnh}}(q,\dot{q},u,t) \end{bmatrix}$$

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, suchs 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]
>>> 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()
>>> 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(FL, BL)
>>> 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) x (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]])
```

The forcing vector can be linearized as well; its Jacobian is taken only with respect to the independent coordinates and speeds. The linearized forcing vector is of size (n + o) x (n - l + o - m), where l is the number of configuration constraints and m is the number of motion constraints. Two matrices are returned; the first is an "A" matrix, or the Jacobian with respect to the independent states, the second is a "B" matrix, or the Jacobian with respect to 'forces'; this can be an empty matrix if there are no 'forces'. Forces here are undefined functions of time (dynamic symbols); they are only allowed to be in the forcing vector and their derivatives are not allowed to be present. If dynamic symbols appear in the mass matrix or kinematic differential equations, an error with be raised.

```
>>> KM.linearize()[0]
Matrix([
[0, 1],
[0, 0]])
```

Exploration of the provided examples is encouraged in order to gain more understanding of the `KanesMethod` object.

Lagrange's Method in Physics/Mechanics

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.

$$\begin{aligned}
\mathbf{m_c}(q,t)\dot{q} + \mathbf{f_c}(q,t) &= 0 \\
\mathbf{m_{dc}}(\dot{q},q,t)\ddot{q} + \mathbf{f_{dc}}(\dot{q},q,t) &= 0 \\
\mathbf{m_d}(\dot{q},q,t)\ddot{q} + \square_{\mathbf{c}}(q,t)\lambda + \mathbf{f_d}(\dot{q},q,t) &= 0
\end{aligned}$$

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:

$$\begin{aligned}
x &= \begin{bmatrix} \ddot{q} \\ \lambda \end{bmatrix} \\
\mathbf{M}(q,t) &= \begin{bmatrix} \mathbf{m_d}(q,t) & \square_{\mathbf{c}}(q,t) \end{bmatrix} \\
\mathbf{F}(\dot{q},q,t) &= \begin{bmatrix} \mathbf{f_d}(q,\dot{q},t) \end{bmatrix}
\end{aligned}$$

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()
>>> 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 in a list of expressions which are equal to zero. It should be noted that `mechanics` requires that the holonomic constraint equations must be supplied as velocity level constraint equations i.e. the holonomic constraint equations must be supplied after they have been differentiated with respect to time. Modifying the example above, the equations of motion can then be generated:

```
>>> LM = LagrangesMethod(L, [q1, q2], coneqs = [q1d - q2d])
```

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) x (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.

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()
```

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 Fr from the force list and Fr* 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(ForceList, BodyList)
>>> 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 1629) for a more thorough explanation of this. Here, we will turn on the automatic simplifcation 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()
>>> 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 an 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(ForceList, BodyList)
>>> 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()
>>> 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.set_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.simplify(); le
Matrix([
[m*r**2*(12*sin(q2)*q3'' + 10*sin(2*q2)*q1'*q2' + 12*cos(q2)*q2'*q3' - 5*cos(2*q2)*q1'' + 7*q1'')/8]
[                     m*r*(8*g*sin(q2) - 5*r*sin(2*q2)*q1'**2 - 12*r*cos(q2)*q1'*q3' + 10*r*q2'')/8]
```

```
[                                                   3*m*r**2*(sin(q2)*q1'' + cos(q2)*q1'*q2' + q3'')/2]
>>> lrhs = l.rhs(); lrhs.simplify(); lrhs
Matrix([
[                                                                  q1'],
[                                                                  q2'],
[                                                                  q3'],
[                      -4*(tan(q2)*q1' + 3*q3'/(2*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 1629). 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()
```

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.

```
>>> WFrad, WRrad, htangle, forkoffset = symbols('WFrad WRrad 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 posses a 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', WFrad*(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 - framelengt
>>> 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*(u
>>> 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 - framelengt
>>> 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 - framelengt
>>> WF_cont.v2pt_theory(WF_mc, N, WF)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) + u4)*Frame.x - framelengt
```

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 nonholonomic 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, q3],
...           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(FL, BL)
>>> 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 = {
...       WFrad: PaperRadFront,
...       WRrad: PaperRadRear,
...       htangle: HTA,
...       forkoffset: rake,
...       forklength: 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 udot = forcing, where MM is the mass matrix, udot 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 xdot = 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 is the referenced paper. This concludes the bicycle example.

Potential Issues/Advanced Topics/Future Features in Physics/Mechanics 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.

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

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.

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.

Substitution Substitution into large expressions can be slow, and take a few minutes.

Linearization Currently, the `Kane` object's `linearize` method doesn'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 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.mechanics import ReferenceFrame, mprint, mpprint, mlatex
>>> N = ReferenceFrame('N', indices=['i', 'j', 'k'])
>>> N['i']
N['i']
>>> N.x
N['i']
>>> mlatex(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.mechanics import ReferenceFrame, mlatex
>>> N = ReferenceFrame('N', latexs=['n1','\mathbf{n}_2','cat'])
>>> mlatex(N.x)
'n1'
>>> mlatex(N.y)
'\\mathbf{n}_2'
>>> mlatex(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.mechanics import dynamicsymbols, mprint
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> dynamicsymbols._t = symbols('T')
>>> q2 = dynamicsymbols('q2')
>>> q2
q2(T)
>>> q1
q1(t)
>>> q1d = dynamicsymbols('q1', 1)
>>> mprint(q1d)
q1'
>>> dynamicsymbols._str = 'd'
>>> mprint(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.

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.

Additional Options on Initialization of Kane, RigidBody, and Particle This would allow a user to specify all relevant information using keyword arguments when creating these objects. This is fairly clear for `RigidBody` and `Point`. For `Kane`, everything but the force and body lists will be able to be entered, as computation of Fr and Fr* can take a while, and produce an output.

Additional Methods for RigidBody and Particle For `RigidBody` and `Particle` (not all methods for `Particle` though), add methods for getting: momentum, angular momentum, and kinetic energy. Additionally, adding a attribute and method for defining potential energy would allow for a total energy method/property.

Also possible is including the method which creates a transformation matrix for 3D animations; this would require a "reference orientation" for a camera as well as a "reference point" for distance to the camera. Development of this could also be tied into code output.

References for Physics/Mechanics

Mechanics API

Essential Components (Docstrings)

CoordinateSym

ReferenceFrame

class `sympy.physics.mechanics.essential.`**`ReferenceFrame`**(*name*, *indices=None*, *latexs=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: ^N alpha ^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.mechanics 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: ^N omega ^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.mechanics 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
```

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.mechanics 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', and 'Axis'. See examples for correct usage.

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.mechanics import ReferenceFrame, Vector
>>> from sympy import symbols
>>> q0, q1, q2, q3, q4 = symbols('q0 q1 q2 q3 q4')
>>> 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], 'XYX')
```

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])
```

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.

```
>>> B.orient(N, 'Axis', [q1, N.x + 2 * N.y])
```

orientnew(*newname, rot_type, amounts, rot_order=‘‘, 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 **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’, and ‘Axis’. See examples for correct usage.

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.mechanics import ReferenceFrame, Vector
>>> from sympy import symbols
>>> q0, q1, q2, q3, q4 = symbols('q0 q1 q2 q3 q4')
>>> 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], 'XYX')
```

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])
```

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.

```
>>> B.orient(N, 'Axis', [q1, N.x + 2 * 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.mechanics 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.mechanics 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
```

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.mechanics.essential.**Vector**(*inlist*)

The class used to define vectors.

It along with ReferenceFrame are the building blocks of describing a classical mechanics system in PyDy.

Attributes

simp	Boolean	Let certain methods use trigsimp on their outputs

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.mechanics 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
```

diff(*wrt, otherframe*)

Takes the partial derivative, with respect to a value, in a frame.

Returns a Vector.

Parameters **wrt** : Symbol

What the partial derivative is taken with respect to.

otherframe : ReferenceFrame

The ReferenceFrame that the partial derivative is taken in.

Examples

```
>>> from sympy.physics.mechanics import ReferenceFrame, Vector, dynamicsymbols
>>> from sympy import Symbol
>>> 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
```

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.mechanics import ReferenceFrame, Vector, 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 the time derivative of the Vector in a ReferenceFrame.

Returns a Vector which is the time derivative of the self Vector, taken in frame otherframe.

Parameters **otherframe** : ReferenceFrame

The ReferenceFrame that the partial derivative is taken in.

Examples

```
>>> from sympy.physics.mechanics import ReferenceFrame, Vector, 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)
>>> A.x.dt(N) == 0
True
>>> v.dt(N)
u1'*N.x
```

express(*otherframe*)

Returns a vector, expressed in the other frame.

A new Vector is returned, equalivalent to this Vector, but its components are all defined in only the otherframe.

Parameters **otherframe** : ReferenceFrame

The frame for this Vector to be described in

Examples

```
>>> from sympy.physics.mechanics 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
```

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.mechanics import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

simplify()

Simplify the elements in the Vector in place.

subs(**args, **kwargs*)

Substituion on the Vector.

Examples

```
>>> from sympy.physics.mechanics import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = N.x * s
>>> a.subs({s: 2})
2*N.x
```

Dyadic

class sympy.physics.mechanics.essential.**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.

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.mechanics 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.mechanics 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.

Parameters **frame** : ReferenceFrame

The frame to take the time derivative in

Examples

```
>>> from sympy.physics.mechanics 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.

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.mechanics 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()

Simplify the elements in the Dyadic in-place.

subs(**args*, ***kwargs*)

Substituion on the Dyadic.

Examples

```
>>> from sympy.physics.mechanics 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)
```

Related Essential Functions (Docstrings)

dynamicsymbols

sympy.physics.mechanics.essential.**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.mechanics import dynamicsymbols :

>>> from sympy import diff, Symbol :

>>> q1 = dynamicsymbols('q1') :

>>> q1 :

q1(t) :

>>> diff(q1, Symbol('t')) :

Derivative(q1(t), t) :

dot

`sympy.physics.mechanics.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.mechanics import ReferenceFrame, Vector, 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.mechanics.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.mechanics 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.mechanics.functions.**outer**(*vec1, vec2*)

Outer product convenience wrapper for Vector.outer(): Returns a vector, expressed in the other frame.

A new Vector is returned, equalivalent to this Vector, but its components are all defined in only the otherframe.

Parameters **otherframe** : ReferenceFrame

The frame for this Vector to be described in

Examples

```
>>> from sympy.physics.mechanics 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
```

express

sympy.physics.mechanics.functions.**express**(*vec, frame, frame2=None*)

Express convenience wrapper for Vector.express(): Returns a vector, expressed in the other frame.

A new Vector is returned, equalivalent to this Vector, but its components are all defined in only the otherframe.

Parameters **otherframe** : ReferenceFrame

The frame for this Vector to be described in

Examples

```
>>> from sympy.physics.mechanics 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
```

time_derivative

get_motion_params

Kinematics (Docstrings)

Point

class sympy.physics.mechanics.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.

a1pt_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 x r^OP + ^N omega^B x (^N omega^B x r^OP) + 2 ^N omega^B x ^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.mechanics import Point, ReferenceFrame, 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.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 x r^OP + ^N omega^B x (^N omega^B x 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.mechanics 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.mechanics 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.mechanics import ReferenceFrame, Point
>>> N = ReferenceFrame('N')
>>> P1 = Point('P1')
>>> P2 = P1.locatenew('P2', 10 * N.x)
```

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.mechanics 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 **value** : Vector

The vector value of this point's acceleration in the frame

frame : ReferenceFrame

The frame in which this point's acceleration is defined

Examples

```
>>> from sympy.physics.mechanics 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 **value** : Vector

The vector which defines the location of this point

point : Point

The other point which this point's location is defined relative to

Examples

```
>>> from sympy.physics.mechanics 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 **value** : Vector

The vector value of this point's velocity in the frame

frame : ReferenceFrame

The frame in which this point's velocity is defined

Examples

```
>>> from sympy.physics.mechanics 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 v^P = ^B v^P + ^N v^O + ^N omega^B x 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.mechanics import Point, ReferenceFrame, 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 v^P = ^N v^O + ^N omega^B x 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.mechanics 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.mechanics 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.mechanics.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: speed[0]*B.x + speed[1]*B.y + 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.mechanics import dynamicsymbols
>>> from sympy.physics.mechanics import kinematic_equations, mprint
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> mprint(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(
```

Masses, Inertias & Particles, RigidBodys (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 = r x m * v

where r is the position vector from point O to the particle P, m is the mass of the particle, and 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
```

get_mass()

Mass of the particle.

get_point()

Point of the particle.

kinetic_energy(*frame*)

Kinetic energy of the particle

The kinetic energy, T, of a particle, P, is given by

'T = 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.set_potential_energy(m * g * h)
>>> P.potential_energy
g*h*m
```

set_potential_energy(*scalar*)

Used to set the potential energy of the Particle.

Parameters **scalar** : Sympifyable

The potential energy (a scalar) 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.set_potential_energy(m * g * h)
```

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)
>>> 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*)

Angular momentum of the rigid body.

The angular momentum H, about some point O, of a rigid body B, in a frame N is given by

H = I* . omega + r* x (M * v)

where I* is the central inertia dyadic of B, omega 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 point O 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)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, Inertia_tuple)
>>> 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, omega 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)
>>> 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.set_potential_energy(M * g * h)
>>> B.potential_energy
M*g*h
```

set_potential_energy(*scalar*)

Used to set the potential energy of this RigidBody.

Parameters **scalar: Sympifyable** :

The potential energy (a scalar) of the RigidBody.

Examples

```
>>> from sympy.physics.mechanics import Particle, Point, outer
>>> from sympy.physics.mechanics import RigidBody, ReferenceFrame
>>> from sympy import symbols
>>> b = ReferenceFrame('b')
>>> M, g, h = symbols('M g h')
>>> P = Point('P')
>>> I = outer (b.x, b.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, Inertia_tuple)
>>> B.set_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 realtive 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 kinetic energy 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 linear momentum of the particle, H1, and the linear 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 kinetic energy 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)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> angular_momentum(O, 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, V1, and the potential energy of the rigid body, V2, i.e.

V = V1 + V2

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.set_potential_energy(m * g * h)
>>> A.set_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 kinetic energy 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.set_potential_energy(m * g * h)
>>> A.set_potential_energy(M * g * h)
>>> Lagrangian(N, Pa, A)
-M*g*h - g*h*m + 350
```

Kane's Method, Lagrange's Method & Associated Functions (Docstrings)

Kane

class sympy.physics.mechanics.kane.**KanesMethod**(*frame, q_ind, u_ind, kd_eqs=None, q_dependent=[], configuration_constraints=[], u_dependent=[], velocity_constraints=[], acceleration_constraints=None, u_auxiliary=[]*)

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] udot = 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
>>> 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 rearrnge them though, so we calculate the mass matrix and the forcing terms, for E.o.M. in the form: [MM] udot = forcing, where MM is the mass matrix, udot 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(FL, BL)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> rhs
Matrix([[(-c*u(t) - k*q(t))/m]])
>>> KM.linearize()[0]
Matrix([
[ 0,  1],
[-k, -c]])
```

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

auxiliary	Matrix	If applicable, the set of auxiliary Kane's equations used to solve for non-contributing forces.
mass_matrix	Matrix	The system's mass matrix
forcing	Matrix	The system's forcing vector
mass_matrix_full	Matrix	The "mass matrix" for the u's and q's
forcing_full	Matrix	The "forcing vector" for the u's and q's

kanes_equations(*FL*, *BL*)

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 **FL** : list

Takes in a list of (Point, Vector) or (ReferenceFrame, Vector) tuples which represent the force at a point or torque on a frame.

BL : list

A list of all RigidBody's and Particle's in the system.

kindiffdict()

Returns the qdot's in a dictionary.

linearize()

Method used to generate linearized equations.

Note that for linearization, it is assumed that time is not perturbed, but only coordinates and positions. The "forcing" vector's jacobian is computed with respect to the state vector in the form [Qi, Qd, Ui, Ud]. This is the "f_lin_A" matrix.

It also finds any non-state dynamicsymbols and computes the jacobian of the "forcing" vector with respect to them. This is the "f_lin_B" matrix; if this is empty, an empty matrix is created.

Consider the following: If our equations are: [M]qudot = f, where [M] is the full mass matrix, qudot is a vector of the deriatives of the coordinates and speeds, and f in the full forcing vector, the linearization process is as follows: [M]qudot = [f_lin_A]qu + [f_lin_B]y, where qu is the state vector, f_lin_A is the jacobian of the full forcing vector with respect to the state vector, f_lin_B is the jacobian of the full forcing vector with respect to any non-speed/coordinate dynamicsymbols which show up in the full forcing vector, and y is a vector of those dynamic symbols (each column in f_lin_B corresponds to a row of the y vector, each of which is a non-speed/coordinate dynamicsymbol).

To get the traditional state-space A and B matrix, you need to multiply the f_lin_A and f_lin_B matrices by the inverse of the mass matrix. Caution needs to be taken when inverting large symbolic matrices; substituting in numerical values before inverting will work better.

A tuple of (f_lin_A, f_lin_B, other_dynamicsymbols) is returned.

rhs(*inv_method=None*)

Returns the system's equations of motion in first order form.

The output of this will be the right hand side of:

[qdot, udot].T = f(q, u, t)

Or, the equations of motion in first order form. 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.

partial_velocity

sympy.physics.mechanics.functions.**partial_velocity**(*vel_list*, *u_list*, *frame*)

Returns a list of partial velocities.

For a list of velocity or angular velocity vectors the partial derivatives with respect to the supplied generalized speeds are computed, in the specified ReferenceFrame.

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_list** : list

List of velocities of Point's and angular velocities of ReferenceFrame's

u_list : list

List of independent generalized speeds.

frame : ReferenceFrame

The ReferenceFrame the partial derivatives are going to be taken in.

Examples

```
>>> from sympy.physics.mechanics import Point, ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.mechanics import partial_velocity
>>> u = dynamicsymbols('u')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> vel_list = [P.vel(N)]
>>> u_list = [u]
>>> partial_velocity(vel_list, u_list, N)
[[N.x]]
```

LagrangesMethod

class sympy.physics.mechanics.lagrange.**LagrangesMethod**(*Lagrangian, q_list, coneqs=None, forcelist=None, frame=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 a list of the generalized coordinates, at the bare minimum. If there are any constraint equations, they can be supplied as keyword arguments. The Lagrangian multipliers are automatically generated and are equal in number to the constraint equations.Similarly any non-conservative forces can be supplied in a list (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 its derivative. 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, a list of nonconservative forces/torques must be constructed, where each entry in is a (Point, Vector) or (ReferenceFrame, Vector) tuple, where the Vectors represent the nonconservative force or torque.

```
>>> Pa = Particle('Pa', P, m)
>>> Pa.set_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 an the Lagrangian, the list of generalized coordinates. Also supplied are the constraint equations, the forcelist and the inertial frame, 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, t)]])
```

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

mass_matrix	Matrix	The system's mass matrix
forcing	Matrix	The system's forcing vector
mass_matrix_full	Matrix	The "mass matrix" for the qdot's, qdoubledot's, and the lagrange multipliers (lam)
forcing_full	Matrix	The forcing vector for the qdot's, qdoubledot's and lagrange multipliers (lam)

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.

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(*method='GE'*)

Returns equations that can be solved numerically

Parameters **method** : string

The method by which matrix inversion of mass_matrix_full must be performed such as Gauss Elimination or LU decomposition.

Printing (Docstrings)

mechanics_printing

sympy.physics.mechanics.functions.**mechanics_printing**()

Sets up interactive printing for mechanics' derivatives.

The main benefit of this is for printing of time derivatives; instead of displaying as Derivative(f(t),t), it will display f' This is only actually needed for when derivatives are present and are not in a physics.mechanics object.

Examples

```
>>> # 2 lines below are for tests to function properly
>>> import sys
>>> sys.displayhook = sys.__displayhook__
>>> from sympy import Function, Symbol, diff
>>> from sympy.physics.mechanics import mechanics_printing
>>> f = Function('f')
>>> t = Symbol('t')
>>> x = Symbol('x')
>>> diff(f(t), t)
Derivative(f(t), t)
>>> mechanics_printing()
>>> diff(f(t), t)
f'
>>> diff(f(x), x)
Derivative(f(x), x)
>>> # 2 lines below are for tests to function properly
>>> import sys
>>> sys.displayhook = sys.__displayhook__
```

mprint

sympy.physics.mechanics.functions.**mprint**(*expr, **settings*)

Function for printing of expressions generated in mechanics.

Extends SymPy's StrPrinter; mprint is equivalent to: print sstr() mprint takes the same options as sstr.

Parameters **expr** : valid sympy object

SymPy expression to print

settings : args

Same as print for SymPy

Examples

```
>>> from sympy.physics.mechanics import mprint, dynamicsymbols
>>> u1 = dynamicsymbols('u1')
>>> print(u1)
u1(t)
>>> mprint(u1)
u1
```

mpprint

sympy.physics.mechanics.functions.**mpprint**(*expr, **settings*)

Function for pretty printing of expressions generated in mechanics.

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 pretty print

Examples

Use in the same way as pprint

mlatex

sympy.physics.mechanics.functions.**mlatex**(*expr, **settings*)

Function for printing latex representation of mechanics 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.mechanics import mlatex, ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> q1dd, q2dd = dynamicsymbols('q1 q2', 2)
>>> mlatex(N.x + N.y)
'\\mathbf{\\hat{n}_x} + \\mathbf{\\hat{n}_y}'
>>> mlatex(q1 + q2)
'q_{1} + q_{2}'
>>> mlatex(q1d)
'\\dot{q}_{1}'
>>> mlatex(q1 * q2d)
'q_{1} \\dot{q}_{2}'
>>> mlatex(q1dd * q1 / q1d)
'\\frac{q_{1} \\ddot{q}_{1}}{\\dot{q}_{1}}'
```

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 [R212] (page 1629) as: `{A, B} = A*B + B*A`. This class returns the anticommutator in an unevaluated form. To evaluate the anticommutator, use the `.doit()` method.

Cannonical 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

[R212] (page 1629)

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 it 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 [R213] (page 1629):

$$C^{j_1,m_1}_{j_2,m_2,j_3,m_3} = \langle j_1, m_1; j_2, m_2 | j_3, m_3 \rangle$$

Parameters **j1, m1, j2, m2, j3, m3** : Number, Symbol

Terms determining the angular momentum of coupled angular momentum systems.

See Also:

Wigner3j (page 1512) Wigner-3j symbols

References

[R213] (page 1629)

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 [R214] (page 1629).

Parameters **j1, m1, j2, m2, j3, m3** : Number, Symbol

Terms determining the angular momentum of coupled angular momentum systems.

See Also:

`CG` **(page 1511)** Clebsch-Gordan coefficients

References

[R214] (page 1629)

Examples

Declare a Wigner-3j coefficient and calcualte 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 1512)** Wigner-3j symbols

class sympy.physics.quantum.cg.**Wigner9j**

Class for the Wigner-9j symbols

See Also:

`Wigner3j` **(page 1512)** 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 [R215] (page 1629).

See Also:

CG (page 1511) Clebsh-Gordan coefficients

References

[R215] (page 1629)

Examples

Simplify the sum over CG(a,alpha,0,0,a,alpha) for all alpha 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 [R216] (page 1630) as: `[A, B] = A*B - B*A`. This class returns the commutator in an unevaluated form. To evaluate the commutator, use the `.doit()` method.

Cannonical 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

[R216] (page 1630)

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 it 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 [R217] (page 1630). For matrices this operation is equivalent to transpose and complex conjugate [R218] (page 1630).

Parameters **arg** : Expr

The sympy expression that we want to take the dagger of.

References

[R217] (page 1630), [R218] (page 1630)

Examples

Daggering 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
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

[R221] (page 1630)

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 a inner products 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 <b|k> 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)
>>> 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 `Tensor-`

Products, 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
AxB*CxD
>>> 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.**XOp**
1D cartesian position operator.

class sympy.physics.quantum.cartesian.**YOp**
Y cartesian coordinate operator (for 2D or 3D systems)

class sympy.physics.quantum.cartesian.**ZOp**
Z cartesian coordinate operator (for 3D systems)

class sympy.physics.quantum.cartesian.**PxOp**
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 [R219] (page 1630).

References

[R219] (page 1630)

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([0, oo))
>>> hs.dimension
oo
>>> hs.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 [R220] (page 1630). This is a mess, so we have a class to represent it directly.

References

[R220] (page 1630)

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, AntiCommutator, represent, apply_operators.

class `sympy.physics.quantum.operator.`**`Operator`**

Base class for non-commuting quantum operators.

An operator maps between quantum states [R222] (page 1630). In quantum mechanics, observables (including, but not limited to, measured physical values) are represented as Hermitian operators [R223] (page 1630).

Parameters **args** : tuple

The list of numbers or parameters that uniquely specify the operator.
For time-dependent operators, this will include the time.

References

[R222] (page 1630), [R223] (page 1630)

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
```

Polymonials 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 == 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*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.**OuterProduct**

An unevaluated outer product between a ket and bra.

This constructs an outer product between any subclass of `KetBase` and `BraBase` as `|a><b|`. An `OuterProduct` inherits from Operator as they act as operators in quantum expressions. For reference see [R224] (page 1630).

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

[R224] (page 1630)

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 arbitary 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.

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 definining the `_represent_SzOp` 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_SzOp(self, basis, **options):
...         return Matrix([1,0])
...
>>> class SzOp(Operator):
...     pass
...
>>> sz = SzOp('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 XOp, XKet, XBra
>>> X = XOp()
>>> 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. <x'|x>) 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 <x'|A|x> 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><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 momemtum.

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 [R225] (page 1630).

Parameters **alpha** : Number, Symbol

First Euler Angle

beta : Number, Symbol

Second Euler angle

gamma : Number, Symbol

Third Euler angle

See Also:

`WignerD` **(page 1534)** Symbolic Wigner-D function

`D` **(page 1532)** Wigner-D function

`d` **(page 1533)** Wigner small-d function

References

[R225] (page 1630)

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 1534)** 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 1534)** 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:

$$< j, m|\mathcal{R}(\alpha, \beta, \gamma)|j', m' >= \delta_{jj'} D(j, m, m', \alpha, \beta, \gamma)$$

Where the rotation operator is as defined by the Rotation class [R226] (page 1630).

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 1532)** Rotation operator

References

[R226] (page 1630)

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 1535) 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 1535) 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 1535) 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 1535) 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 1538)** Coupled eigenstates

TensorProduct Used to specify uncoupled states

uncouple **(page 1541)** Uncouples states given coupling parameters

couple **(page 1540)** Couples uncoupled states

Examples

Normal States:

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKet, JxKet
>>> from sympy import symbols
>>> JzKet(1, 0)
|1,0>
>>> j, m = symbols('j m')
>>> 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),JxKet(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],
[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 1535) 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 1538) 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 1538) 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 1538) 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 1538) 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 in `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 indicies of the product spaces that are considered to be coupled together. For example, if we want to couple j_1 and j_4, the indicies 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 argular 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 1535)** Normal spin eigenstates

`uncouple` **(page 1541)** Uncoupling of coupling spin states

`couple` **(page 1540)** 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 1538)** 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
```

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 construt 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 [R227] (page 1630).

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

[R227] (page 1630)

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
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 [R228] (page 1630). 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

[R228] (page 1630)

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
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 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

```
>>> 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)
```

Attributes

nargs	

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 `(-oo, oo)`.

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, (x, 0, 1))
>>> 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!

class sympy.physics.quantum.circuitplot.**CircuitPlot**(*c, nqubits, **kwargs*)

A class for managing a circuit plot.

control_line(*gate_idx*, *min_wire*, *max_wire*)
Draw a vertical control line.

control_point(*gate_idx*, *wire_idx*)
Draw a control point.

not_point(*gate_idx*, *wire_idx*)
Draw a NOT gates as the circle with plus in the middle.

one_qubit_box(*t*, *gate_idx*, *wire_idx*)
Draw a box for a single qubit gate.

swap_point(*gate_idx*, *wire_idx*)
Draw a swap point as a cross.

update(*kwargs*)
Load the kwargs into the instance dict.

sympy.physics.quantum.circuitplot.**circuit_plot**(*c*, *nqubits*, ***kwargs*)
Draw the circuit diagram for the circuit with nqubits.

Parameters **c** : circuit

The circuit to plot. Should be a product of Gate instances.

nqubits : int

The number of qubits to include in the circuit. Must be at least as big as the largest min_qubits' of the gates.

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.

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>` and does nothing if the state is `|0>`.

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> and does nothing if the state is |0>.

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 CNotGate (page 1553)

sympy.physics.quantum.gate.**SWAP**

alias of SwapGate (page 1553)

sympy.physics.quantum.gate.**H**

alias of HadamardGate (page 1552)

sympy.physics.quantum.gate.**X**

alias of XGate (page 1552)

sympy.physics.quantum.gate.**Y**

alias of YGate (page 1552)

`sympy.physics.quantum.gate.`**`Z`**
alias of ZGate (page 1552)

`sympy.physics.quantum.gate.`**`T`**
alias of TGate (page 1552)

`sympy.physics.quantum.gate.`**`S`**
alias of PhaseGate (page 1552)

`sympy.physics.quantum.gate.`**`Phase`**
alias of PhaseGate (page 1552)

`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.

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>` 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><phi| - 1` on some qubits. `|phi> = (tensor product of n Hadamards)*(|0> with n 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 Rk 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 QTF.

`sympy.physics.quantum.qft.`**`Rk`**

alias of `RkGate` (page 1557)

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>` 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>` 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 1557)** 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'*)

Coverts 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 specifed 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.**continued_fraction**(*x*, *y*)

This applies the continued fraction expansion to two numbers x/y

x is the numerator and y is the denominator

```
>>> from sympy.physics.quantum.shor import continued_fraction
>>> continued_fraction(3, 8)
[0, 2, 1, 2]
```

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>|0>` 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.

5.37 Category Theory Module

5.37.1 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.

5.37.2 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 1564).

It is prohibited to instantiate this class. Use one of the derived classes instead.

See Also:

`IdentityMorphism` (page 1566), `NamedMorphism` (page 1564), `CompositeMorphism` (page 1565)

`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 1563)

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")
>>> 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 1563) 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")
>>> 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 1564) with the supplied name, otherwise returns a `Morphism` (page 1563). 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")
>>> (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 1563)

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 $\mathrm{K} = (O, \mathrm{hom}, id, \circ)$ consisting of

- a (set-theoretical) class O, whose members are called K-objects,
- for each pair (A, B) of K-objects, a set $\mathrm{hom}(A, B)$ whose members are called K-morphisms from A to B,
- for a each K-object A, a morphism $id : A \to A$, called the K-identity of A,
- a composition law $\circ$ associating with every K-morphisms $f : A \to B$ and $g : B \to C$ a K-morphism $g \circ f : A \to C$, called the composite of f and g.

Composition is associative, K-identities are identities with respect to composition, and the sets $\mathrm{hom}(A, B)$ 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 1568) can be asserted to be commutative in a `Category` (page 1567) by supplying the argument `commutative_diagrams` in the constructor.

See Also:

`Diagram` (page 1568)

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 1568) 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
>>> 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 1563), Morphism (page 1563)

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)
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")
>>> d = Diagram([f, g], {f: "unique", g*f: "veryunique"})
>>> d1 = d.subdiagram_from_objects(FiniteSet(A, B))
>>> d1 == Diagram([f], {f: "unique"})
True
```

5.37.3 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 moprhisms. 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 1571) using keyword arguments. The currently supported hints are explained in the following paragraphs.

`DiagramGrid` (page 1571) 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 1571) provides a hint for that purpose:

```
>>> grid = DiagramGrid(diagram, layout="sequential", transpose=True)
>>> pprint(grid)
A

B

C

D

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")
>>> 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)[]_{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 1576)

Notes

Instances of ArrowStringDescription (page 1574) will be constructed by XypicDiagramDrawer (page 1576) and provided for further use in formatters. The user is not expected to construct instances of ArrowStringDescription (page 1574) 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://www.tug.org/applications/Xy-pic/

class sympy.categories.diagram_drawing.**XypicDiagramDrawer**

Given a Diagram and the corresponding DiagramGrid (page 1571), 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 1571):

```
>>> 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 is accepts an ArrowStringDescription (page 1574) 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 ∃!, 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} \\
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 1578), `ArrowStringDescription` (page 1574)

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
>>> 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 1571):

```
>>> 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 1571) and `XypicDiagramDrawer` (page 1576). 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 1571).

For more information about the arguments, see the docstrings of DiagramGrid (page 1571) and `XypicDiagramDrawer.draw`.

See Also:

XypicDiagramDrawer (page 1576), DiagramGrid (page 1571)

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[ld]^{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)
```

5.38 Differential Geometry Module

5.38.1 Introduction

5.38.2 Base Class Reference

class sympy.diffgeom.**Manifold**(*name, dim*)

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**(*name, manifold*)

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 parametrization 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**(*name, patch, names=None*)

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
>>> 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])
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.**BaseScalarField**(*coord_sys, index*)
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 unappropriate 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 filed:

```
>>> 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**(*coord_sys, index*)

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 wrt 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 unappropriate 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))||
\dxi_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))||
\dxi_2                          /|xi_2=r0*sin(theta0)
```

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))||
\dxi_1            /|xi_1=x
>>> pprint(dg(e_y))
/  d              \|
|-----(g(x, xi_2))||
\dxi_2            /|xi_2=y
```

Applying the exterior derivative operator twice always results in:

```
>>> Differential(dg)
0
```

class sympy.diffgeom.**BaseCovarDerivativeOp**(*coord_sys, index, christoffel*)

Covariant derivative operator wrt 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)
>>> 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
```

5.39 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).

CHAPTER

SIX

DEVELOPMENT TIPS: COMPARISONS IN PYTHON

6.1 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.

6.2 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))
```

6.3 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 1590) 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.

6.4 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....

6.5 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):
        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__()
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)
```

CHAPTER

SEVEN

WIKI

SymPy has a public wiki located at http://wiki.sympy.org. Users should feel free to contribute to this wiki anything interesting/useful.

7.1 FAQ

FAQ is one of the most useful wiki pages. It has answers to frequently-asked questions.

CHAPTER

EIGHT

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.

CHAPTER

NINE

PLANET SYMPY

We have a blog aggregator at http://planet.sympy.org.

CHAPTER

TEN

SYMPY LOGOS

SymPy has a collection of official logos, which can be generated from $sympy.svg < https : //github.com/sympy/sympy/tree/master/doc/src/sympy.svg >$ 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.

CHAPTER

ELEVEN

PROJECTS USING SYMPY

This is an (incomplete) list of projects that use SymPy. If you use SymPy in your project, please let us know on our mailinglist, so that we can add your project here as well.

- SfePy (simple finite elements in Python)
- Quameon (Quantum Monte Carlo in Python)

CHAPTER

TWELVE

BLOGS, NEWS, MAGAZINES

You can see a list of blogs/news/magazines mentioning SymPy (that we know of) on our wiki page.

CHAPTER

THIRTEEN

ABOUT

13.1 SymPy Development Team

SymPy is a team project and it was developed by a lot of people.

Here is a list of contributors together with what they do, (and in some cases links to their wiki pages), where they describe in more details what they do and what they are interested in (some people didn't want to be mentioned here, so see our repository history for a full list).

1. Ondřej Čertík: started the project in 2006, on Jan 4, 2011 passed the project leadership to Aaron Meurer
2. Fabian Pedregosa: everything, reviewing patches, releases, general advice (issues and mailinglist), GSoC 2009
3. Jurjen N.E. Bos: pretty printing and other patches
4. Mateusz Paprocki: GSoC 2007, concrete math module, integration module, new core integration, a lot of patches, general advice, new polynomial module, improvements to solvers, simplifications, patch review
5. Marc-Etienne M.Leveille: matrix patch
6. Brian Jorgensen: GSoC 2007, plotting module and related things, patches
7. Jason Gedge: GSoC 2007, geometry module, a lot of patches and fixes, new core integration
8. Robert Schwarz: GSoC 2007, polynomials module, patches
9. Pearu Peterson: new core, sympycore project, general advice (issues and mailinglist)
10. Fredrik Johansson: mpmath project and its integration in SymPy, number theory, combinatorial functions, products & summation, statistics, units, patches, documentation, general advice (issues and mailinglist)
11. Chris Wu: GSoC 2007, linear algebra module
12. Ulrich Hecht: pattern matching and other patches
13. Goutham Lakshminarayan: number theory functions
14. David Lawrence: GHOP, Mathematica parser, square root denesting
15. Jaroslaw Tworek: GHOP, sympify AST implementation, sqrt() refactoring, maxima parser and other patches
16. David Marek: GHOP, derivative evaluation patch, int(NumberSymbol) fix
17. Bernhard R. Link: documentation patch

18. Andrej Tokarčík: GHOP, python printer
19. Or Dvory: GHOP, documentation
20. Saroj Adhikari: bug fixes
21. Pauli Virtanen: bug fix
22. Robert Kern: bug fix, common subexpression elimination
23. James Aspnes: bug fixes
24. Nimish Telang: multivariate lambdas
25. Abderrahim Kitouni: pretty printing + complex expansion bug fixes
26. Pan Peng: ode solvers patch
27. Friedrich Hagedorn: many bug fixes, refactorings and new features added
28. Elrond der Elbenfuerst: pretty printing fix
29. Rizgar Mella: BBP formula for pi calculating algorithm
30. Felix Kaiser: documentation + whitespace testing patches
31. Roberto Nobrega: several pretty printing patches
32. David Roberts: latex printing patches
33. Sebastian Krämer: implemented lambdify/numpy/mpmath cooperation, bug fixes, refactoring, lambdifying of matrices, large printing refactoring and bugfixes
34. Vinzent Steinberg: docstring patches, a lot of bug fixes, nsolve (nonlinear equation systems solver), compiling functions to machine code, patches review
35. Riccardo Gori: improvements and speedups to matrices, many bug fixes
36. Case Van Horsen: implemented optional support for gmpy in mpmath
37. Štěpán Roučka: a lot of bug fixes all over SymPy (matrix, simplification, limits, series, ...)
38. Ali Raza Syed: pretty printing/isympy on windows fix
39. Stefano Maggiolo: many bug fixes, polishings and improvements
40. Robert Cimrman: matrix patches
41. Bastian Weber: latex printing patches
42. Sebastian Krause: match patches
43. Sebastian Kreft: latex printing patches, Dirac delta function, other fixes
44. Dan (coolg49964): documentation fixes
45. Alan Bromborsky: geometric algebra modules
46. Boris Timokhin: matrix fixes
47. Robert (average.programmer): initial piecewise function patch
48. Andy R. Terrel: piecewise function polishing and other patches
49. Hubert Tsang: LaTeX printing fixes
50. Konrad Meyer: policy patch
51. Henrik Johansson: matrix zero finder patch
52. Priit Laes: line integrals, cleanups in ODE, tests added
53. Freddie Witherden: trigonometric simplifications fixes, LaTeX printer improvements

54. Brian E. Granger: second quantization physics module
55. Andrew Straw: lambdify() improvements
56. Kaifeng Zhu: factorint() fix
57. Ted Horst: Basic.__call__() fix, pretty printing fix
58. Andrew Docherty: Fix to series
59. Akshay Srinivasan: printing fix, improvements to integration
60. Aaron Meurer: ODE solvers (GSoC 2009), The Risch Algorithm for integration (GSoC 2010), other fixes, project leader as of Jan 4, 2011
61. Barry Wardell: implement series as function, several tests added
62. Tomasz Buchert: ccode printing fixes, code quality concerning exceptions, documentation
63. Vinay Kumar: polygamma tests
64. Johann Cohen-Tanugi: commutative diff tests
65. Jochen Voss: a test for the @cachit decorator and several other fixes
66. Luke Peterson: improve solve() to handle Derivatives
67. Chris Smith: improvements to solvers, many bug fixes, documentation and test improvements
68. Thomas Sidoti: MathML printer improvements
69. Florian Mickler: reimplementation of convex_hull, several geometry module fixes
70. Nicolas Pourcelot: Infinity comparison fixes, latex fixes
71. Ben Goodrich: Matrix.jacobian() function improvements and fixes
72. Toon Verstraelen: code generation module, latex printing improvements
73. Ronan Lamy: test coverage script; limit, expansion and other fixes and improvements; cleanup
74. James Abbatiello: fixes tests on windows
75. Ryan Krauss: fixes could_extract_minus_sign(), latex fixes and improvements
76. Bill Flynn: substitution fixes
77. Kevin Goodsell: Fix to Integer/Rational
78. Jorn Baayen: improvements to piecewise functions and latex printing, bug fixes
79. Eh Tan: improve trigonometric simplification
80. Renato Coutinho: derivative improvements
81. Oscar Benjamin: latex printer fix, gcd bug fix
82. Øyvind Jensen: implemented coupled cluster expansion and wick theorem, improvements to assumptions, bugfixes
83. Julio Idichekop Filho: indentation fixes, docstring improvements
84. Łukasz Pankowski: fix matrix multiplication with numpy scalars
85. Chu-Ching Huang: fix 3d plotting example
86. Fernando Perez: symarray() implemented
87. Raffaele De Feo: fix non-commutative expansion
88. Christian Muise: fixes to logic module

89. Matt Curry: GSoC 2010 project (symbolic quantum mechanics)
90. Kazuo Thow: cleanup pretty printing tests
91. Christian Schubert: Fix to sympify()
92. Jezreel Ng: fix hyperbolic functions rewrite
93. James Pearson: Py3k related fixes
94. Matthew Brett: fixes to lambdify
95. Addison Cugini: GSoC 2010 project (quantum computing)
96. Nicholas J.S. Kinar: improved documentation about "Immutability of Expressions"
97. Harold Erbin: Geometry related work
98. Thomas Dixon: fix a limit bug
99. Cristóvão Sousa: implements _sage_ method for sign function.
100. Andre de Fortier Smit: doctests in matrices improved
101. Mark Dewing: Fixes to Integral/Sum, MathML work
102. Alexey U. Gudchenko: various work in matrices
103. Gary Kerr: fix examples, docs
104. Sherjil Ozair: fixes to SparseMatrix
105. Oleksandr Gituliar: fixes to Matrix
106. Sean Vig: Quantum improvement
107. Prafullkumar P. Tale: fixed plotting documentation
108. Vladimir Perić: fix some Python 3 issues
109. Tom Bachmann: fixes to Real
110. Yuri Karadzhov: improvements to hyperbolic identities
111. Vladimir Lagunov: improvements to the geometry module
112. Matthew Rocklin: stats, sets, matrix expressions
113. Saptarshi Mandal: Test for an integral
114. Gilbert Gede: Tests for solvers
115. Anatolii Koval: Infinite 1D box example
116. Tomo Lazovich: fixes to quantum operators
117. Pavel Fedotov: fix to is_number
118. Kibeom Kim: fixes to cse function and preorder_traversal
119. Gregory Ksionda: fixes to Integral instantiation
120. Tomáš Bambas: prettier printing of examples
121. Jeremias Yehdegho: fixes to the nthoery module
122. Jack McCaffery: fixes to asin and acos
123. Raymond Wong: Quantum work
124. Luca Weihs: improvements to the geometry module
125. Shai 'Deshe' Wyborski: fix to numeric evaluation of hypergeometric sums
126. Thomas Wiecki: Fix Sum.diff

127. Óscar Nájera: better Laguerre polynomial generator
128. Mario Pernici: faster algorithm for computing Groebner bases
129. Benjamin McDonald: Fix bug in geometry
130. Sam Magura: Improvements to Plot.saveimage
131. Stefan Krastanov: Make Pyglet an external dependency
132. Bradley Froehle: Fix shell command to generate modules in setup.py
133. Min Ragan-Kelley: Fix isympy to work with IPython 0.11
134. Nikhil Sarda: Fix to combinatorics/prufer
135. Emma Hogan: Fixes to the documentation
136. Jason Moore: Fixes to the mechanics module
137. Julien Rioux: Fix behavior of some deprecated methods
138. Roberto Colistete, Jr.: Add num_columns option to the pretty printer
139. Raoul Bourquin: Implement Euler numbers
140. Gert-Ludwig Ingold: Correct derivative of coth
141. Srinivas Vasudevan: Implementation of Catalan numbers
142. Miha Marolt: Add numpydoc extension to the Sphinx docs, fix ode_order
143. Tim Lahey: Rotation matrices
144. Luis Garcia: update examples in symarry
145. Matt Rajca: Code quality fixes
146. David Li: Documentation fixes
147. David Ju: Increase test coverage, fixes to KroneckerDelta
148. Alexandr Gudulin: Code quality fixes
149. Bilal Akhtar: isympy man page
150. Grzegorz Świrski: Fix to latex(), MacPorts portfile
151. Matt Habel: SymPy-wide pyflakes editing
152. Nikolay Lazarov: Translation of the tutorial to Bulgarian
153. Nichita Utiu: Add pretty printing to Product()
154. Tristan Hume: Fixes to test_lambdify.py
155. Imran Ahmed Manzoor: Fixes to the test runner
156. Steve Anton: pyflakes cleanup of various modules, documentation fixes for logic
157. Sam Sleight: Fixes to geometry documentation
158. tsmars15: Fixes to code quality
159. Chancellor Arkantos: Fixes to the logo
160. Stepan Simsa: Translation of the tutorial to Czech
161. Tobias Lenz: Unicode pretty printer for Sum
162. Siddhanathan Shanmugam: Documentation fixes for the Physics module
163. Tiffany Zhu: Improved the latex() docstring
164. Alexey Subach: Translation of the tutorial to Russian

165. Joan Creus: Improvements to the test runner
166. Geoffry Song: Improve code coverage
167. Puneeth Chaganti: Fix for the tutorial
168. Marcin Kostrzewa: SymPy Cheatsheet
169. Jim Zhang: Fixes to AUTHORS and .mailmap
170. Natalia Nawara: Fixes to Product()
171. vishal: Update the Czech tutorial translation to use a .po file
172. Shruti Mangipudi: added See Also documentation to Matrices
173. Davy Mao: Documentation
174. Swapnil Agarwal: added See Also documentation to ntheory, functions
175. Kendhia: Add XFAIL tests
176. jerryma1121: See Also documentation for geometry
177. Joachim Durchholz: corrected spacing issues identified by PyDev
178. Martin Povišer: fix problem with rationaltools
179. Siddhant Jain: See Also documentation for functions/special
180. Kevin Hunter: Improvements to the inequality classes
181. Michael Mayorov: Improvement to series
182. Nathan Alison: Additions to the stats module
183. Christian Bühler: remove use of set_main() in GA
184. Carsten Knoll: improvement to preview()
185. M R Bharath: modified use of int_tested, improvement to Permutation
186. Matthias Toews: File permissions
187. Sergiu Ivanov: Fixes to zoo, SymPyDeprecationWarning
188. Jorge E. Cardona: Cleanup in polys
189. Sanket Agarwal: Rewrite coverage_doctest.py script
190. Manoj Babu K.: Improve gamma function
191. Sai Nikhil: Fix to Heavyside with complex arguments
192. Aleksandar Makelov: Fixes regarding the dihedral group generator
193. Raphael Michel: German translation of the tutorial
194. Sachin Irukula: Changes to allow Dict sorting
195. Ashwini Oruganti: Changes to Pow printing
196. Andreas Kloeckner: Fix to cse()
197. Prateek Papriwal: improve summation documentation
198. Arpit Goyal: Improvements to Integral and Sum
199. Angadh Nanjangud: in physics.mechanics, added a function and tests for the parallel axis theorem
200. Comer Duncan: added dual, is_antisymmetric, and det_lu_decomposition to matrices.py
201. Jens H. Nielsen: added sets to modules listing, update IPython printing extension

202. Joseph Dougherty: modified whitespace cleaning to remove multiple newlines at eof
203. marshall2389: Spelling correction
204. Guru Devanla: Implemented quantum density operator
205. George Waksman: Implemented JavaScript code printer and MathML printer
206. Angus Griffith: Fix bug in rsolve
207. Timothy Reluga: Rewrite trigonometric functions as rationals
208. Brian Stephanik: Test for a bug in fcode
209. Ljubiša Moćić: Serbian translation of the tutorial
210. Piotr Korgul: Polish translation of the tutorial
211. Rom le Clair: French translation of the tutorial
212. Alexandr Popov: Fixes to Pauli algebra
213. Saurabh Jha: Work on Kauers algorithm
214. Tarun Gaba: Implemented some trigonometric integrals
215. Takafumi Arakaki: Add info target to the doc Makefile
216. Alexander Eberspächer: correct typo in aboutus.rst
217. Sachin Joglekar: Simplification of logic expressions to SOP and POS forms
218. Tyler Pirtle: Fix improperly formatted error message
219. Vasily Povalyaev: Fix latex(Min)
220. Colleen Lee: replace uses of fnan with S.NaN
221. Niklas Thörne: Fix links in the docs
222. Huijun Mai: Chinese translation of the tutorial
223. Marek Šuppa: Improvements to symbols, tests
224. Prasoon Shukla: Bug fixes
225. Sergey B Kirpichev: Bug fixes
226. Stefen Yin: Fixes to the mechanics module
227. Thomas Hisch: Improvements to the printing module
228. Matthew Hoff: Addition to quantum module
229. Madeleine Ball: Bug fix
230. Case Van Horsen: Fixes to gmpy support
231. Mary Clark: Improvements to the group theory module
232. Rishabh Dixit: Bug fixes
233. Acebulf: Typos
234. Manoj Kumar: Bug fix
235. Akshit Agarwal: improvements to range handling in symbols
236. CJ Carey: Fix for limits of factorials
237. Patrick Lacasse: Fix for Piecewise.subs
238. Ananya H: Bug fix
239. Tarang Patel: added test for issue 1640

240. Christopher Dembia: improvements to mecahanics documentation
241. Benjamin Fishbein: added rank method to Matrix
242. Sean Ge: made KroneckerDelta arguments canonically ordered
243. Ankit Agrawal: Statistical moments
244. Amit Jamadagni: qapply Rotation to spin states
245. Björn Dahlgren: Documentation fix
246. Christophe Saint-Jean: fixed and added metrics to galgebra
247. Demian Wassermann: fix to ccode printer for Piecewise
248. Khagesh Patel: Addition to matrix expressions
249. Stephen Loo: Update minimum gmpy2 version
250. hm: Fixes to printing
251. Katja Sophie Hotz: use expansion in minpoly
252. Varun Joshi: Addition to functions
253. Chetna Gupta: Improvements to the Risch integration algorithm
254. Thilina Rathnayake: Fix to the matrices
255. Shravas K Rao: Implement prev_lexicographic and next_lexicographic
256. Max Hutchinson: Fix to HadamardProduct
257. Matthew Tadd: fix definition in units module
258. Alexander Hirzel: Updates to ODE docs
259. Randy Heydon: improve collinear point detection
260. Ramana Venkata: improvements to special functions
261. Oliver Lee: improvements to mechanics
262. Seshagiri Prabhu: hardcoded 3x3 determinant
263. Pradyumna: Fix to printing
264. Erik Welch: Fix a warning
265. Eric Nelson: Fixes to printing
266. Roland Puntaier: Improve App Engine support
267. Chris Conley: Use warnings instead of prints
268. Tim Swast: Help with pull requests and IPython
269. Dmitry Batkovich: Fix to series
270. Francesco Bonazzi: Improvements to matrices and tensors
271. Yuriy Demidov: Add examples from “Review of CAS mathematical capabilities”
272. Rick Muller: Implementation of quantum circuit plotting
273. Manish Gill: Fix infinite loop in Matrix constructor
274. Markus Müller: Add Jordan form for matrices
275. Amit Saha: Fixes to documentation
276. Jeremy: Bug fix
277. QuaBoo: Optimizations in ntheory

278. Stefan van der Walt: Fixes to mechanics module
279. David Joyner: Cryptography module
280. Lars Buitinck: Bug fix
281. Alkiviadis G. Akritas: Add a reference
282. Vinit Ravishankar: fix iterables documentation
283. Mike Boyle: Additions to the printing system
284. Heiner Kirchhoffer: PythonRational int conversion fix
285. Pablo Puente: Test cases from Wester paper
286. James Fiedler: Bug fix
287. Harsh Gupta: Bug fix
288. Tuomas Airaksinen: is_real for besselx
289. rathmann: fix some Python 2/3 issues
290. Paul Strickland: documentation fix
291. James Goppert: correct code in form_lagranges_equations

Up-to-date list in the order of the first contribution is given in the AUTHORS file.

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)
- 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)
- 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)

- 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) 2006-2013 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:

  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 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.
```

BIBLIOGRAPHY

[R19] 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/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. Currently, the bool of the –Than operators will give True or False arbitrarily. 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. (True) and (y > z)
6. (y > z)
7. LessThanObject

Because of the "and" added at step 2, the statement gets turned into a weak ternary statement. If the first object evalutes __nonzero__ as True, then the second object, (y > z) is returned. If the first object evaluates __nonzero__ as False (step 5), then (x > y) is returned.

> 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 is an open PEP to change this, **PEP 335**, but until that is implemented, this cannot be made to work.

[R20] For more information, see these two bug reports:

"Separate boolean and symbolic relationals" Issue 1887

"It right 0 < x < 1 ?" Issue 2960

[R21] 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/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. Currently, the bool of the –Than operators will give True or False arbitrarily. 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. (True) and (y > z)
6. (y > z)
7. LessThanObject

Because of the "and" added at step 2, the statement gets turned into a weak ternary statement. If the first object evalutes __nonzero__ as True, then the second object, (y > z) is returned. If the first object evaluates __nonzero__ as False (step 5), then (x > y) is returned.

> 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 is an open PEP to change this, **PEP 335**, but until that is implemented, this cannot be made to work.

[R22] For more information, see these two bug reports:

"Separate boolean and symbolic relationals" Issue 1887

"It right 0 < x < 1 ?" Issue 2960

[R23] 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/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. Currently, the bool of the –Than operators will give True or False arbitrarily. 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. (True) and (y > z)
6. (y > z)
7. LessThanObject

Because of the "and" added at step 2, the statement gets turned into a weak ternary statement. If the first object evalutes __nonzero__ as True, then the second object, (y > z) is returned. If the first object evaluates __nonzero__ as False (step 5), then (x > y) is returned.

> 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 is an open PEP to change this, **PEP 335**, but until that is implemented, this cannot be made to work.

[R24] For more information, see these two bug reports:

"Separate boolean and symbolic relationals" Issue 1887

"It right 0 < x < 1 ?" Issue 2960

[R25] 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/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. Currently, the bool of the –Than operators will give True or False arbitrarily. 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. (True) and (y > z)
6. (y > z)
7. LessThanObject

Because of the "and" added at step 2, the statement gets turned into a weak ternary statement. If the first object evalutes __nonzero__ as True, then the second object, (y > z) is returned. If the first object evaluates __nonzero__ as False (step 5), then (x > y) is returned.

> 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 is an open PEP to change this, **PEP 335**, but until that is implemented, this cannot be made to work.

[R26] For more information, see these two bug reports:

"Separate boolean and symbolic relationals" Issue 1887

"It right 0 < x < 1 ?" Issue 2960

[R4] Skiena, S. 'Permutations.' 1.1 in Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 3-16, 1990.

[R5] Knuth, D. E. The Art of Computer Programming, Vol. 4: Combinatorial Algorithms, 1st ed. Reading, MA: Addison-Wesley, 2011.

[R6] 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

[R7] D. L. Kreher, D. R. Stinson 'Combinatorial Algorithms' CRC Press, 1999

[R8] Graham, R. L.; Knuth, D. E.; and Patashnik, O. Concrete Mathematics: A Foundation for Computer Science, 2nd ed. Reading, MA: Addison-Wesley, 1994.

[R9] http://en.wikipedia.org/wiki/Permutation#Product_and_inverse

[R10] http://en.wikipedia.org/wiki/Lehmer_code

[R11] http://mathworld.wolfram.com/LabeledTree.html

[R12] 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

[R13] http://en.wikipedia.org/wiki/Summation#Capital-sigma_notation

[R14] http://en.wikipedia.org/wiki/Empty_sum

[R15] 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

[R16] http://en.wikipedia.org/wiki/Multiplication#Capital_Pi_notation

[R17] http://en.wikipedia.org/wiki/Empty_product

[R18] Marko Petkovsek, Herbert S. Wilf, Doron Zeilberger, A = B, AK Peters, Ltd., Wellesley, MA, USA, 1997, pp. 73–100

[R51] http://planetmath.org/encyclopedia/DefinitionsInTrigonometry.html

[R52] http://en.wikipedia.org/wiki/Directed_complete_partial_order

[R53] http://en.wikipedia.org/wiki/Lattice_%28order%29

[R54] http://planetmath.org/encyclopedia/DefinitionsInTrigonometry.html

[R55] http://planetmath.org/encyclopedia/DefinitionsInTrigonometry.html

[R27] http://en.wikipedia.org/wiki/Bell_number

[R28] http://mathworld.wolfram.com/BellNumber.html

[R29] http://mathworld.wolfram.com/BellPolynomial.html

[R30] http://en.wikipedia.org/wiki/Bernoulli_number

[R31] http://en.wikipedia.org/wiki/Bernoulli_polynomial

[R32] http://mathworld.wolfram.com/BernoulliNumber.html

[R33] http://mathworld.wolfram.com/BernoulliPolynomial.html

[R34] http://en.wikipedia.org/wiki/Catalan_number

[R35] http://mathworld.wolfram.com/CatalanNumber.html

[R36] http://functions.wolfram.com/GammaBetaErf/CatalanNumber/

[R37] http://geometer.org/mathcircles/catalan.pdf

[R38] http://en.wikipedia.org/wiki/Euler_numbers

[R39] http://mathworld.wolfram.com/EulerNumber.html

[R40] http://en.wikipedia.org/wiki/Alternating_permutation

[R41] http://mathworld.wolfram.com/AlternatingPermutation.html

[R42] http://en.wikipedia.org/wiki/Fibonacci_number

[R43] http://mathworld.wolfram.com/FibonacciNumber.html

[R44] http://en.wikipedia.org/wiki/Harmonic_number

[R45] http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/

[R46] http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/

[R47] http://en.wikipedia.org/wiki/Lucas_number

[R48] http://mathworld.wolfram.com/LucasNumber.html

[R49] http://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind

[R50] http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind

[R56] http://mathworld.wolfram.com/DeltaFunction.html

[R57] http://mathworld.wolfram.com/HeavisideStepFunction.html

[R58] http://en.wikipedia.org/wiki/Gamma_function

[R59] http://mathworld.wolfram.com/LogGammaFunction.html

[R60] http://en.wikipedia.org/wiki/Polygamma_function

[R61] http://functions.wolfram.com/GammaBetaErf/PolyGamma/

[R62] http://functions.wolfram.com/GammaBetaErf/PolyGamma2/

[R63] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables

[R64] http://en.wikipedia.org/wiki/Incomplete_gamma_function

[R65] http://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions

[R66] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables

[R67] http://en.wikipedia.org/wiki/Incomplete_gamma_function

[R68] http://dlmf.nist.gov/6.6

[R69] http://en.wikipedia.org/wiki/Exponential_integral

[R70] Abramowitz & Stegun, section 5: http://www.math.sfu.ca/~cbm/aands/page_228.htm

[R71] http://dlmf.nist.gov/8.19

[R72] http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/

[R73] http://en.wikipedia.org/wiki/Exponential_integral

[R74] http://en.wikipedia.org/wiki/Logarithmic_integral

[R75] http://mathworld.wolfram.com/LogarithmicIntegral.html

[R76] http://dlmf.nist.gov/6

[R77] http://mathworld.wolfram.com/SoldnersConstant.html

[R78] http://en.wikipedia.org/wiki/Logarithmic_integral

[R79] http://mathworld.wolfram.com/LogarithmicIntegral.html

[R80] http://dlmf.nist.gov/6

[R81] http://en.wikipedia.org/wiki/Trigonometric_integral

[R82] http://en.wikipedia.org/wiki/Trigonometric_integral

[R83] http://en.wikipedia.org/wiki/Trigonometric_integral

[R84] http://en.wikipedia.org/wiki/Trigonometric_integral

[R85] http://en.wikipedia.org/wiki/Fresnel_integral

[R86] http://dlmf.nist.gov/7

[R87] http://mathworld.wolfram.com/FresnelIntegrals.html

[R88] http://functions.wolfram.com/GammaBetaErf/FresnelS

[R89] http://en.wikipedia.org/wiki/Fresnel_integral

[R90] http://dlmf.nist.gov/7

[R91] http://mathworld.wolfram.com/FresnelIntegrals.html

[R92] http://functions.wolfram.com/GammaBetaErf/FresnelC

[R93] http://en.wikipedia.org/wiki/Error_function

[R94] http://dlmf.nist.gov/7

[R95] http://mathworld.wolfram.com/Erf.html

[R96] http://functions.wolfram.com/GammaBetaErf/Erf

[R97] http://en.wikipedia.org/wiki/Error_function

[R98] http://dlmf.nist.gov/7

[R99] http://mathworld.wolfram.com/Erfc.html

[R100] http://functions.wolfram.com/GammaBetaErf/Erfc

[R101] http://en.wikipedia.org/wiki/Error_function

[R102] http://mathworld.wolfram.com/Erfi.html

[R103] http://functions.wolfram.com/GammaBetaErf/Erfi

[R104] http://functions.wolfram.com/GammaBetaErf/Erf2/

[R105] http://en.wikipedia.org/wiki/Error_function#Inverse_functions

[R106] http://functions.wolfram.com/GammaBetaErf/InverseErf/

[R107] http://en.wikipedia.org/wiki/Error_function#Inverse_functions

[R108] http://functions.wolfram.com/GammaBetaErf/InverseErfc/

[R109] http://functions.wolfram.com/GammaBetaErf/InverseErf2/

[R110] Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 9", Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables

[R111] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1

[R112] http://en.wikipedia.org/wiki/Bessel_function

[R113] http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/

[R114] http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/

[R115] http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/

[R116] http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/

[R117] http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/

[R118] http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/

[R119] http://en.wikipedia.org/wiki/B-spline

[R120] http://dlmf.nist.gov/25.11

[R121] http://en.wikipedia.org/wiki/Hurwitz_zeta_function

[R122] http://en.wikipedia.org/wiki/Dirichlet_eta_function

[R123] Bateman, H.; Erdelyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill. Section 1.11.

[R124] http://dlmf.nist.gov/25.14

[R125] http://en.wikipedia.org/wiki/Lerch_transcendent

[R126] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1

[R127] http://en.wikipedia.org/wiki/Generalized_hypergeometric_function

[R128] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1

[R129] http://en.wikipedia.org/wiki/Meijer_G-function

[R130] http://en.wikipedia.org/wiki/Elliptic_integrals
[R131] http://functions.wolfram.com/EllipticIntegrals/EllipticK
[R132] http://en.wikipedia.org/wiki/Elliptic_integrals
[R133] http://functions.wolfram.com/EllipticIntegrals/EllipticF
[R134] http://en.wikipedia.org/wiki/Elliptic_integrals
[R135] http://functions.wolfram.com/EllipticIntegrals/EllipticE2
[R136] http://functions.wolfram.com/EllipticIntegrals/EllipticE
[R137] http://en.wikipedia.org/wiki/Elliptic_integrals
[R138] http://functions.wolfram.com/EllipticIntegrals/EllipticPi3
[R139] http://functions.wolfram.com/EllipticIntegrals/EllipticPi
[R140] http://en.wikipedia.org/wiki/Jacobi_polynomials
[R141] http://mathworld.wolfram.com/JacobiPolynomial.html
[R142] http://functions.wolfram.com/Polynomials/JacobiP/
[R143] http://en.wikipedia.org/wiki/Jacobi_polynomials
[R144] http://mathworld.wolfram.com/JacobiPolynomial.html
[R145] http://functions.wolfram.com/Polynomials/JacobiP/
[R146] http://en.wikipedia.org/wiki/Gegenbauer_polynomials
[R147] http://mathworld.wolfram.com/GegenbauerPolynomial.html
[R148] http://functions.wolfram.com/Polynomials/GegenbauerC3/
[R149] http://en.wikipedia.org/wiki/Chebyshev_polynomial
[R150] http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html
[R151] http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html
[R152] http://functions.wolfram.com/Polynomials/ChebyshevT/
[R153] http://functions.wolfram.com/Polynomials/ChebyshevU/
[R154] http://en.wikipedia.org/wiki/Chebyshev_polynomial
[R155] http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html
[R156] http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html
[R157] http://functions.wolfram.com/Polynomials/ChebyshevT/
[R158] http://functions.wolfram.com/Polynomials/ChebyshevU/
[R159] http://en.wikipedia.org/wiki/Legendre_polynomial
[R160] http://mathworld.wolfram.com/LegendrePolynomial.html
[R161] http://functions.wolfram.com/Polynomials/LegendreP/
[R162] http://functions.wolfram.com/Polynomials/LegendreP2/
[R163] http://en.wikipedia.org/wiki/Associated_Legendre_polynomials
[R164] http://mathworld.wolfram.com/LegendrePolynomial.html
[R165] http://functions.wolfram.com/Polynomials/LegendreP/
[R166] http://functions.wolfram.com/Polynomials/LegendreP2/
[R167] http://en.wikipedia.org/wiki/Hermite_polynomial

[R168] http://mathworld.wolfram.com/HermitePolynomial.html

[R169] http://functions.wolfram.com/Polynomials/HermiteH/

[R170] http://en.wikipedia.org/wiki/Laguerre_polynomial

[R171] http://mathworld.wolfram.com/LaguerrePolynomial.html

[R172] http://functions.wolfram.com/Polynomials/LaguerreL/

[R173] http://functions.wolfram.com/Polynomials/LaguerreL3/

[R174] http://en.wikipedia.org/wiki/Laguerre_polynomial#Assoc_laguerre_polynomials

[R175] http://mathworld.wolfram.com/AssociatedLaguerrePolynomial.html

[R176] http://functions.wolfram.com/Polynomials/LaguerreL/

[R177] http://functions.wolfram.com/Polynomials/LaguerreL3/

[R178] http://en.wikipedia.org/wiki/Spherical_harmonics

[R179] http://mathworld.wolfram.com/SphericalHarmonic.html

[R180] http://functions.wolfram.com/Polynomials/SphericalHarmonicY/

[R181] http://dlmf.nist.gov/14.30

[R182] http://en.wikipedia.org/wiki/Spherical_harmonics

[R183] http://mathworld.wolfram.com/SphericalHarmonic.html

[R184] http://functions.wolfram.com/Polynomials/SphericalHarmonicY/

[R185] http://en.wikipedia.org/wiki/Spherical_harmonics

[R186] http://mathworld.wolfram.com/SphericalHarmonic.html

[R187] http://functions.wolfram.com/Polynomials/SphericalHarmonicY/

[R188] http://en.wikipedia.org/wiki/Kronecker_delta

[Doran] http://www.mrao.cam.ac.uk/~cjld1/pages/book.htm `Geometric Algebra for Physicists` by C. Doran and A. Lasenby, Cambridge University Press, 2003.

[Hestenes] http://geocalc.clas.asu.edu/html/CA_to_GC.html `Clifford Algebra to Geometric Calculus` by D.Hestenes and G. Sobczyk, Kluwer Academic Publishers, 1984.

[Macdonald] '<http://faculty.luther.edu/~macdonal>'_ `Linear and Geometric Algebra` by Alan Macdonald, http://www.amazon.com/Alan-Macdonald/e/B004MB2QJQ

[WikiPappus] "Pappus's Hexagon Theorem" Wikipedia, the Free Encyclopedia. Web. 26 Apr. 2013. <http://en.wikipedia.org/wiki/Pappus's_hexagon_theorem>

[BlogPost] http://nessgrh.wordpress.com/2011/07/07/tricky-branch-cuts/

[Bro05] M. Bronstein, Symbolic Integration I: Transcendental Functions, Second Edition, Springer-Verlag, 2005, pp. 35-70

[R189] http://en.wikipedia.org/wiki/Gaussian_quadrature

[R190] http://people.sc.fsu.edu/~jburkardt/cpp_src/legendre_rule/legendre_rule.html

[R191] http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature

[R192] http://people.sc.fsu.edu/~jburkardt/cpp_src/laguerre_rule/laguerre_rule.html

[R193] http://en.wikipedia.org/wiki/Gauss-Hermite_Quadrature

[R194] http://people.sc.fsu.edu/~jburkardt/cpp_src/hermite_rule/hermite_rule.html

[R195] http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_hermite_rule/gen_hermite_rule.html

[R196] http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature

[R197] http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_laguerre_rule/gen_laguerre_rule.html

[R198] http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature

[R199] http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev1_rule/chebyshev1_rule.html

[R200] http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature

[R201] http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev2_rule/chebyshev2_rule.html

[R202] http://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature

[R203] http://people.sc.fsu.edu/~jburkardt/cpp_src/jacobi_rule/jacobi_rule.html

[R204] http://people.sc.fsu.edu/~jburkardt/cpp_src/gegenbauer_rule/gegenbauer_rule.html

[R205] en.wikipedia.org/wiki/Quine-McCluskey_algorithm

[R206] en.wikipedia.org/wiki/Quine-McCluskey_algorithm

[AbramowitzStegun] M Abramowitz & I Stegun. *Handbook of Mathematical Functions, 9th Ed.*, Tenth Printing, December 1972, with corrections

[Bailey] D H Bailey. "Tanh-Sinh High-Precision Quadrature", http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-tanh-sinh.pdf

[BenderOrszag] C M Bender & S A Orszag. *Advanced Mathematical Methods for Scientists and Engineers*, Springer 1999

[BorweinBailey] J Borwein, D H Bailey & R Girgensohn. *Experimentation in Mathematics - Computational Paths to Discovery*, A K Peters, 2003

[BorweinBorwein] J Borwein & P B Borwein. *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity*, Wiley 1987

[BorweinZeta] P Borwein. "An Efficient Algorithm for the Riemann Zeta Function", http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P115.pdf

[CabralRosetti] L G Cabral-Rosetti & M A Sanchis-Lozano. "Appell Functions and the Scalar One-Loop Three-point Integrals in Feynman Diagrams". http://arxiv.org/abs/hep-ph/0206081

[Carlson] B C Carlson. "Numerical computation of real or complex elliptic integrals". http://arxiv.org/abs/math/9409227v1

[Corless] R M Corless et al. "On the Lambert W function", Adv. Comp. Math. 5 (1996) 329-359. http://www.apmaths.uwo.ca/~djeffrey/Offprints/W-adv-cm.pdf

[DLMF] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/

[GradshteynRyzhik] I S Gradshteyn & I M Ryzhik, A Jeffrey & D Zwillinger (eds.), *Table of Integrals, Series and Products*, Seventh edition (2007), Elsevier

[GravesMorris] P R Graves-Morris, D E Roberts & A Salam. "The epsilon algorithm and related topics", *Journal of Computational and Applied Mathematics*, Volume 122, Issue 1-2 (October 2000)

[MPFR] The MPFR team. "The MPFR Library: Algorithms and Proofs", http://www.mpfr.org/algorithms.pdf

[Slater] L J Slater. *Generalized Hypergeometric Functions*. Cambridge University Press, 1966

[Spouge] J L Spouge. "Computation of the gamma, digamma, and trigamma functions", SIAM J. Numer. Anal. Vol. 31, No. 3, pp. 931-944, June 1994.

[SrivastavaKarlsson] H M Srivastava & P W Karlsson. *Multiple Gaussian Hypergeometric Series*. Ellis Horwood, 1985.

[Vidunas] R Vidunas. "Identities between Appell's and hypergeometric functions". http://arxiv.org/abs/0804.0655

[Weisstein] E W Weisstein. *MathWorld*. http://mathworld.wolfram.com/

[WhittakerWatson] E T Whittaker & G N Watson. *A Course of Modern Analysis*. 4th Ed. 1946 Cambridge University Press

[Wikipedia] *Wikipedia, the free encyclopedia*. http://en.wikipedia.org

[WolframFunctions] Wolfram Research, Inc. *The Wolfram Functions Site*. http://functions.wolfram.com/

[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 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, http://citeseer.ist.psu.edu/ajwa95grbner.html, 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

[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

[R230] Big O notation

[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.

[R231] http://en.wikipedia.org/wiki/Arcsine_distribution

[R232] http://en.wikipedia.org/wiki/Benini_distribution

[R233] http://reference.wolfram.com/legacy/v8/ref/BeniniDistribution.html

[R234] http://en.wikipedia.org/wiki/Beta_distribution

[R235] http://mathworld.wolfram.com/BetaDistribution.html

[R236] http://en.wikipedia.org/wiki/Beta_prime_distribution

[R237] http://mathworld.wolfram.com/BetaPrimeDistribution.html

[R238] http://en.wikipedia.org/wiki/Cauchy_distribution

[R239] http://mathworld.wolfram.com/CauchyDistribution.html

[R240] http://en.wikipedia.org/wiki/Chi_distribution

[R241] http://mathworld.wolfram.com/ChiDistribution.html

[R242] http://en.wikipedia.org/wiki/Noncentral_chi_distribution

[R243] http://en.wikipedia.org/wiki/Chi_squared_distribution

[R244] http://mathworld.wolfram.com/Chi-SquaredDistribution.html
[R245] http://en.wikipedia.org/wiki/Dagum_distribution
[R246] http://en.wikipedia.org/wiki/Erlang_distribution
[R247] http://mathworld.wolfram.com/ErlangDistribution.html
[R248] http://en.wikipedia.org/wiki/Exponential_distribution
[R249] http://mathworld.wolfram.com/ExponentialDistribution.html
[R250] http://en.wikipedia.org/wiki/F-distribution
[R251] http://mathworld.wolfram.com/F-Distribution.html
[R252] http://en.wikipedia.org/wiki/Fisher%27s_z-distribution
[R253] http://mathworld.wolfram.com/Fishersz-Distribution.html
[R254] http://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution
[R255] http://en.wikipedia.org/wiki/Gamma_distribution
[R256] http://mathworld.wolfram.com/GammaDistribution.html
[R257] http://en.wikipedia.org/wiki/Inverse-gamma_distribution
[R258] http://en.wikipedia.org/wiki/Kumaraswamy_distribution
[R259] http://en.wikipedia.org/wiki/Laplace_distribution
[R260] http://mathworld.wolfram.com/LaplaceDistribution.html
[R261] http://en.wikipedia.org/wiki/Logistic_distribution
[R262] http://mathworld.wolfram.com/LogisticDistribution.html
[R263] http://en.wikipedia.org/wiki/Lognormal
[R264] http://mathworld.wolfram.com/LogNormalDistribution.html
[R265] http://en.wikipedia.org/wiki/Maxwell_distribution
[R266] http://mathworld.wolfram.com/MaxwellDistribution.html
[R267] http://en.wikipedia.org/wiki/Nakagami_distribution
[R268] http://en.wikipedia.org/wiki/Normal_distribution
[R269] http://mathworld.wolfram.com/NormalDistributionFunction.html
[R270] http://en.wikipedia.org/wiki/Pareto_distribution
[R271] http://mathworld.wolfram.com/ParetoDistribution.html
[R272] http://en.wikipedia.org/wiki/U-quadratic_distribution
[R273] http://en.wikipedia.org/wiki/Raised_cosine_distribution
[R274] http://en.wikipedia.org/wiki/Rayleigh_distribution
[R275] http://mathworld.wolfram.com/RayleighDistribution.html
[R276] http://en.wikipedia.org/wiki/Student_t-distribution
[R277] http://mathworld.wolfram.com/Studentst-Distribution.html
[R278] http://en.wikipedia.org/wiki/Triangular_distribution
[R279] http://mathworld.wolfram.com/TriangularDistribution.html
[R280] http://en.wikipedia.org/wiki/Uniform_distribution_%28continuous%29
[R281] http://mathworld.wolfram.com/UniformDistribution.html

[R282] http://en.wikipedia.org/wiki/Uniform_sum_distribution

[R283] http://mathworld.wolfram.com/UniformSumDistribution.html

[R284] http://en.wikipedia.org/wiki/Von_Mises_distribution

[R285] http://mathworld.wolfram.com/vonMisesDistribution.html

[R286] http://en.wikipedia.org/wiki/Weibull_distribution

[R287] http://mathworld.wolfram.com/WeibullDistribution.html

[R288] http://en.wikipedia.org/wiki/Wigner_semicircle_distribution

[R289] http://mathworld.wolfram.com/WignersSemicircleLaw.html

[R207] http://en.wikipedia.org/wiki/Complex_beam_parameter

[R208] http://en.wikipedia.org/wiki/Ray_transfer_matrix_analysis

[R209] http://en.wikipedia.org/wiki/Gamma_matrices

[R210] http://en.wikipedia.org/wiki/Pauli_matrices

[R211] http://en.wikipedia.org/wiki/Pauli_matrices

[R229] 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)

[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.

[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.

[R212] http://en.wikipedia.org/wiki/Commutator

[R213] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.

[R214] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.

[R215] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.

[R216] http://en.wikipedia.org/wiki/Commutator
[R217] http://en.wikipedia.org/wiki/Hermitian_adjoint
[R218] http://en.wikipedia.org/wiki/Hermitian_transpose
[R221] http://en.wikipedia.org/wiki/Inner_product
[R219] http://en.wikipedia.org/wiki/Hilbert_space
[R220] http://en.wikipedia.org/wiki/Fock_space
[R222] http://en.wikipedia.org/wiki/Operator_%28physics%29
[R223] http://en.wikipedia.org/wiki/Observable
[R224] http://en.wikipedia.org/wiki/Outer_product
[R225] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
[R226] Varshalovich, D A, Quantum Theory of Angular Momentum. 1988.
[R227] http://en.wikipedia.org/wiki/Bra-ket_notation
[R228] http://en.wikipedia.org/wiki/Bra-ket_notation

PYTHON MODULE INDEX

m

mpmath.functions.elliptic, 813

s

sympy, 612
sympy.assumptions, 1147
sympy.assumptions.ask, 1147
sympy.assumptions.assume, 1149
sympy.assumptions.handlers, 1151
sympy.assumptions.handlers.calculus, 1151
sympy.assumptions.handlers.ntheory, 1153
sympy.assumptions.handlers.order, 1154
sympy.assumptions.handlers.sets, 1154
sympy.assumptions.refine, 1150
sympy.categories, 1563
sympy.categories.diagram_drawing, 1571
sympy.combinatorics.generators, 200
sympy.combinatorics.graycode, 241
sympy.combinatorics.group_constructs, 254
sympy.combinatorics.named_groups, 246
sympy.combinatorics.partitions, 172
sympy.combinatorics.perm_groups, 201
sympy.combinatorics.permutations, 177
sympy.combinatorics.polyhedron, 228
sympy.combinatorics.prufer, 231
sympy.combinatorics.subsets, 235
sympy.combinatorics.tensor_can, 255
sympy.combinatorics.testutil, 254
sympy.combinatorics.util, 248
sympy.core.add, 131
sympy.core.basic, 84
sympy.core.cache, 84
sympy.core.compatibility, 168
sympy.core.containers, 167
sympy.core.evalf, 167
sympy.core.expr, 97
sympy.core.exprtools, 170
sympy.core.function, 149
sympy.core.mod, 133
sympy.core.mul, 128
sympy.core.multidimensional, 148
sympy.core.numbers, 120
sympy.core.power, 127
sympy.core.relational, 134
sympy.core.sets, 1172
sympy.core.singleton, 97
sympy.core.symbol, 117
sympy.core.sympify, 81
sympy.crypto.crypto, 282
sympy.diffgeom, 1580
sympy.functions, 302
sympy.functions.special.elliptic_integrals, 378
sympy.functions.special.error_functions, 341
sympy.functions.special.polynomials, 381
sympy.functions.special.zeta_functions, 368
sympy.galgebra, 398
sympy.galgebra.debug, 432
sympy.galgebra.ga, 399
sympy.galgebra.manifold, 432
sympy.galgebra.ncutil, 432
sympy.galgebra.precedence, 432
sympy.galgebra.printing, 432
sympy.galgebra.stringarrays, 432
sympy.galgebra.vector, 432
sympy.geometry, 432
sympy.geometry.curve, 461
sympy.geometry.ellipse, 464
sympy.geometry.entity, 436
sympy.geometry.line, 446
sympy.geometry.point, 440
sympy.geometry.polygon, 477
sympy.geometry.util, 438
sympy.integrals, 498
sympy.integrals.meijerint_doc, 519
sympy.integrals.transforms, 499

sympy.logic, 541
sympy.matrices, 547
sympy.matrices.expressions, 614
sympy.matrices.expressions.blockmatrix, 618
sympy.matrices.immutable, 612
sympy.matrices.matrices, 547
sympy.matrices.sparse, 601
sympy.ntheory.factor_, 266
sympy.ntheory.generate, 261
sympy.ntheory.modular, 275
sympy.ntheory.multinomial, 277
sympy.ntheory.partitions_, 279
sympy.ntheory.primetest, 279
sympy.ntheory.residue_ntheory, 280
sympy.physics, 1382
sympy.physics.gaussopt, 1382
sympy.physics.hep.gamma_matrices, 1426
sympy.physics.hydrogen, 1392
sympy.physics.matrices, 1395
sympy.physics.mechanics.functions, 1492
sympy.physics.mechanics.kane, 1503
sympy.physics.mechanics.lagrange, 1506
sympy.physics.mechanics.particle, 1492
sympy.physics.mechanics.point, 1487
sympy.physics.mechanics.rigidbody, 1495
sympy.physics.paulialgebra, 1396
sympy.physics.qho_1d, 1396
sympy.physics.quantum.anticommutator, 1510
sympy.physics.quantum.cartesian, 1518
sympy.physics.quantum.cg, 1511
sympy.physics.quantum.circuitplot, 1549
sympy.physics.quantum.commutator, 1513
sympy.physics.quantum.constants, 1514
sympy.physics.quantum.dagger, 1514
sympy.physics.quantum.gate, 1550
sympy.physics.quantum.grover, 1555
sympy.physics.quantum.hilbert, 1519
sympy.physics.quantum.innerproduct, 1515
sympy.physics.quantum.operator, 1521
sympy.physics.quantum.operatorset, 1525
sympy.physics.quantum.piab, 1562
sympy.physics.quantum.qapply, 1527
sympy.physics.quantum.qft, 1557
sympy.physics.quantum.qubit, 1557
sympy.physics.quantum.represent, 1528
sympy.physics.quantum.shor, 1561
sympy.physics.quantum.spin, 1532
sympy.physics.quantum.state, 1542
sympy.physics.quantum.tensorproduct, 1516
sympy.physics.secondquant, 1398
sympy.physics.sho, 1397
sympy.physics.units, 1424
sympy.physics.wigner, 1418
sympy.plotting.plot, 1132
sympy.plotting.pygletplot, 1144
sympy.printing.ccode, 1117
sympy.printing.codeprinter, 1127
sympy.printing.conventions, 1127
sympy.printing.fcode, 1118
sympy.printing.gtk, 1121
sympy.printing.lambdarepr, 1121
sympy.printing.latex, 1121
sympy.printing.mathml, 1123
sympy.printing.precedence, 1128
sympy.printing.pretty.pretty, 1116
sympy.printing.pretty.pretty_symbology, 1128
sympy.printing.pretty.stringpict, 1129
sympy.printing.preview, 1126
sympy.printing.printer, 1113
sympy.printing.python, 1124
sympy.printing.repr, 1124
sympy.printing.str, 1124
sympy.printing.tree, 1125
sympy.series, 1164
sympy.sets.fancysets, 1181
sympy.simplify.cse_main, 1200
sympy.simplify.epathtools, 1202
sympy.simplify.hyperexpand, 1201
sympy.simplify.hyperexpand_doc, 1212
sympy.simplify.simplify, 68
sympy.simplify.sqrtdenest, 1200
sympy.simplify.traversaltools, 1201
sympy.solvers, 1295
sympy.solvers.ode, 1284
sympy.solvers.pde, 1294
sympy.solvers.recurr, 1305
sympy.statistics, 1215
sympy.stats, 1222
sympy.stats.crv, 1255
sympy.stats.crv_types, 1256
sympy.stats.frv, 1255
sympy.stats.frv_types, 1256
sympy.stats.rv, 1255
sympy.tensor, 1315
sympy.tensor.index_methods, 1322
sympy.tensor.indexed, 1315
sympy.tensor.tensor, 1325
sympy.utilities, 1335
sympy.utilities.autowrap, 1336
sympy.utilities.codegen, 1340
sympy.utilities.decorator, 1345
sympy.utilities.iterables, 1348
sympy.utilities.lambdify, 1365

sympy.utilities.memoization, 1367
sympy.utilities.misc, 1367
sympy.utilities.pkgdata, 1368
sympy.utilities.pytest, 1369
sympy.utilities.randtest, 1370
sympy.utilities.runtests, 1371
sympy.utilities.source, 1377
sympy.utilities.timeutils, 1378

INDEX

Symbols

_TensorManager (class in sympy.tensor.tensor), 1325
_base_ordering() (in module sympy.combinatorics.util), 248
_check_cycles_alt_sym() (in module sympy.combinatorics.util), 249
_cmp_perm_lists() (in module sympy.combinatorics.testutil), 254
_distribute_gens_by_base() (in module sympy.combinatorics.util), 249
_handle_precomputed_bsgs() (in module sympy.combinatorics.util), 250
_naive_list_centralizer() (in module sympy.combinatorics.testutil), 254
_orbits_transversals_from_bsgs() (in module sympy.combinatorics.util), 251
_print() (sympy.printing.printer.Printer method), 1115
_remove_gens() (in module sympy.combinatorics.util), 251
_strip() (in module sympy.combinatorics.util), 252
_strong_gens_from_distr() (in module sympy.combinatorics.util), 253
_verify_bsgs() (in module sympy.combinatorics.testutil), 254
_verify_centralizer() (in module sympy.combinatorics.testutil), 255
_verify_normal_closure() (in module sympy.combinatorics.testutil), 255

A

A (sympy.physics.gaussopt.RayTransferMatrix attribute), 1388
a (sympy.physics.quantum.shor.CMod attribute), 1562
a1pt_theory() (sympy.physics.mechanics.point.Point method), 1487
a2idx() (in module sympy.matrices.matrices), 595
a2pt_theory() (sympy.physics.mechanics.point.Point method), 1487
AbelianGroup() (in module sympy.combinatorics.named_groups), 248
above() (sympy.printing.pretty.stringpict.stringPict method), 1129
Abs (class in sympy.functions.elementary.complexes), 304
abs() (sympy.polys.domains.domain.Domain method), 1032
abs() (sympy.polys.polyclasses.DMP method), 1044
abs() (sympy.polys.polytools.Poly method), 967
acc() (sympy.physics.mechanics.point.Point method), 1488
accepted_latex_functions (in module sympy.printing.latex), 1121
acos (class in sympy.functions.elementary.trigonometric), 304
acos() (in module mpmath), 677
acosh (class in sympy.functions.elementary.hyperbolic), 305
acosh() (in module mpmath), 682
acot (class in sympy.functions.elementary.trigonometric), 305
acot() (in module mpmath), 679
acoth (class in sympy.functions.elementary.hyperbolic), 305
acoth() (in module mpmath), 683
acsc() (in module mpmath), 679
acsch() (in module mpmath), 682
Add (class in sympy.core.add), 131
add() (sympy.assumptions.assume.AssumptionsContext method), 1149
Add() (sympy.assumptions.handlers.calculus.AskBoundedHa static method), 1152
Add() (sympy.assumptions.handlers.calculus.AskInfinitesima static method), 1153
Add() (sympy.assumptions.handlers.order.AskNegativeHand static method), 1154

Add() (sympy.assumptions.handlers.sets.AskAntiHermitianHandler static method), 1154
Add() (sympy.assumptions.handlers.sets.AskHermitianHandler static method), 1154
Add() (sympy.assumptions.handlers.sets.AskImaginaryHandler static method), 1155
Add() (sympy.assumptions.handlers.sets.AskIntegerHandler static method), 1155
Add() (sympy.assumptions.handlers.sets.AskRationalHandler static method), 1155
Add() (sympy.assumptions.handlers.sets.AskRealHandler static method), 1155
add() (sympy.matrices.matrices.MatrixBase method), 560
add() (sympy.matrices.sparse.SparseMatrix method), 601
add() (sympy.polys.domains.domain.Domain method), 1032
add() (sympy.polys.polyclasses.DMF method), 1049
add() (sympy.polys.polyclasses.DMP method), 1044
add() (sympy.polys.polytools.Poly method), 967
add_ground() (sympy.polys.polyclasses.DMP method), 1044
add_ground() (sympy.polys.polytools.Poly method), 967
adjoint() (sympy.matrices.immutable.ImmutableMatrix method), 613
adjoint() (sympy.matrices.matrices.MatrixBase method), 560
adjugate() (sympy.matrices.matrices.MatrixBase method), 560
agm() (in module mpmath), 671
airyai() (in module mpmath), 744
airyaizero() (in module mpmath), 753
airybi() (in module mpmath), 749
airybizero() (in module mpmath), 754
algebraic_field() (sympy.polys.domains.AlgebraicField method), 1040
algebraic_field() (sympy.polys.domains.domain.Domain method), 1032
algebraic_field() (sympy.polys.domains.IntegerRing method), 1038
algebraic_field() (sympy.polys.domains.RationalField method), 1039
AlgebraicField (class in sympy.polys.domains), 1039
AlgebraicNumber (class in sympy.polys.numberfields), 1004
all_coeffs() (sympy.polys.polyclasses.DMP method), 1044
all_coeffs() (sympy.polys.polytools.Poly method), 968
all_monoms() (sympy.polys.polyclasses.DMP method), 1044
all_monoms() (sympy.polys.polytools.Poly method), 968
all_terms() (sympy.polys.polytools.Poly method), 968
all_terms() (sympy.polys.polyclasses.DMP method), 1044
all_terms() (sympy.polys.polytools.Poly method), 968
allhints (in module sympy.solvers.ode), 1263
almosteq() (in module mpmath), 644
alternating() (sympy.combinatorics.generators static method), 200
AlternatingGroup() (in module sympy.combinatorics.named_groups), 247
altitudes (sympy.geometry.polygon.Triangle attribute), 492
altzeta() (in module mpmath), 839
an (sympy.functions.special.hyper.meijerg attribute), 378
And (class in sympy.logic.boolalg), 543
Anderson (class in mpmath.calculus.optimization), 881
ANewton (class in mpmath.calculus.optimization), 882
ang_acc_in() (sympy.physics.mechanics.essential.ReferenceFrame method), 1475
ang_vel_in() (sympy.physics.mechanics.essential.ReferenceFrame method), 1476
angerj() (in module mpmath), 739
angle (sympy.physics.gaussopt.GeometricRay attribute), 1387
angle_between() (sympy.geometry.line.LinearEntity method), 447
angles (sympy.geometry.polygon.Polygon attribute), 479
angles (sympy.geometry.polygon.RegularPolygon attribute), 485
angular_momentum() (in module sympy.physics.mechanics.functions), 1500
angular_momentum() (sympy.physics.mechanics.particle.Particle method), 1493
angular_momentum() (sympy.physics.mechanics.rigidbody.RigidBody method), 1496
AnnihilateBoson (class in sympy.physics.secondquant), 1403

AnnihilateFermion (class in sympy.physics.secondquant), 1403
annotated() (in module sympy.printing.pretty.pretty_symbology), 1129
ANP (class in sympy.polys.polyclasses), 1049
AntiCommutator (class in sympy.physics.quantum.anticommutator), 1510
AntiSymmetricTensor (class in sympy.physics.secondquant), 1415
any() (in module sympy.parsing.sympy_tokenize), 1380
aother (sympy.functions.special.hyper.meijerg attribute), 378
ap (sympy.functions.special.hyper.hyper attribute), 375
ap (sympy.functions.special.hyper.meijerg attribute), 378
apart() (in module sympy.polys.partfrac), 1007
apart() (sympy.core.expr.Expr method), 97
apart_list() (in module sympy.polys.partfrac), 1007
apery (mpmath.mp attribute), 657
apoapsis (sympy.geometry.ellipse.Ellipse attribute), 466
apothem (sympy.geometry.polygon.RegularPolygon attribute), 485
appellf1() (in module mpmath), 808
appellf2() (in module mpmath), 810
appellf3() (in module mpmath), 811
appellf4() (in module mpmath), 812
append() (sympy.plotting.plot.Plot method), 1134
AppliedPredicate (class in sympy.assumptions.assume), 1149
apply() (sympy.printing.pretty.stringpict.prettyForm static method), 1131
apply() (sympy.simplify.epathtools.EPath method), 1202
apply_grover() (in module sympy.physics.quantum.grover), 1556
apply_operator() (sympy.physics.secondquant.AnnihilateBoson method), 1403
apply_operator() (sympy.physics.secondquant.AnnihilateFermion method), 1404
apply_operator() (sympy.physics.secondquant.CreateBoson method), 1403
apply_operator() (sympy.physics.secondquant.CreateFermion method), 1405
apply_operators() (in module sympy.physics.secondquant), 1407
applyfunc() (sympy.matrices.sparse.SparseMatrix method), 602
approximation() (sympy.core.numbers.NumberSymbol method), 125
arange() (in module mpmath), 648
arbitrary_point() (sympy.geometry.curve.Curve method), 462
arbitrary_point() (sympy.geometry.ellipse.Ellipse method), 466
arbitrary_point() (sympy.geometry.line.LinearEntity method), 447
arbitrary_point() (sympy.geometry.polygon.Polygon method), 479
Arcsin() (in module sympy.stats), 1225
are_similar() (in module sympy.geometry.util), 439
area (sympy.geometry.ellipse.Ellipse attribute), 466
area (sympy.geometry.polygon.Polygon attribute), 480
area (sympy.geometry.polygon.RegularPolygon attribute), 485
arg (class in sympy.functions.elementary.complexes), 306
arg (sympy.assumptions.assume.AppliedPredicate attribute), 1149
arg() (in module mpmath), 641
args (sympy.core.basic.Basic attribute), 85
args (sympy.geometry.polygon.RegularPolygon attribute), 486
args (sympy.polys.polytools.Poly attribute), 969
args (sympy.tensor.indexed.IndexedBase attribute), 1321
args_cnc() (sympy.core.expr.Expr method), 97
Argument (class in sympy.utilities.codegen), 1342
argument (sympy.functions.special.bessel.BesselBase attribute), 362
argument (sympy.functions.special.hyper.hyper attribute), 375
argument (sympy.functions.special.hyper.meijerg attribute), 378
array_form (sympy.combinatorics.permutations.Permutation attribute), 181
array_form (sympy.combinatorics.polyhedron.Polyhedron attribute), 228
ArrowStringDescription (class in sympy.categories.diagram_drawing), 1574
as_base_exp() (sympy.core.function.Function method), 155

as_base_exp() (sympy.core.power.Pow method), 127
as_base_exp() (sympy.functions.elementary.exponential.exp method), 312
as_coeff_Add() (sympy.core.add.Add method), 131
as_coeff_add() (sympy.core.add.Add method), 131
as_coeff_Add() (sympy.core.expr.Expr method), 98
as_coeff_add() (sympy.core.expr.Expr method), 98
as_coeff_Add() (sympy.core.numbers.Number method), 120
as_coeff_exponent() (sympy.core.expr.Expr method), 98
as_coeff_Mul() (sympy.core.expr.Expr method), 98
as_coeff_mul() (sympy.core.expr.Expr method), 98
as_coeff_Mul() (sympy.core.mul.Mul method), 128
as_coeff_Mul() (sympy.core.numbers.Number method), 121
as_coefficient() (sympy.core.expr.Expr method), 99
as_coefficients_dict() (sympy.core.add.Add method), 131
as_coefficients_dict() (sympy.core.expr.Expr method), 100
as_content_primitive() (sympy.core.add.Add method), 131
as_content_primitive() (sympy.core.basic.Basic method), 85
as_content_primitive() (sympy.core.expr.Expr method), 100
as_content_primitive() (sympy.core.mul.Mul method), 128
as_content_primitive() (sympy.core.numbers.Rational method), 124
as_content_primitive() (sympy.core.power.Pow method), 127
as_dict() (sympy.combinatorics.partitions.IntegerPartition method), 174
as_dict() (sympy.polys.polytools.Poly method), 969
as_explicit() (sympy.matrices.expressions.MatrixExpr method), 614
as_expr() (sympy.core.expr.Expr method), 101
as_expr() (sympy.polys.numberfields.AlgebraicNumber method), 1004
as_expr() (sympy.polys.polytools.Poly method), 969
as_ferrers() (sympy.combinatorics.partitions.IntegerPartition method), 174
as_immutable() (sympy.matrices.sparse.SparseMatrix method), 602
as_independent() (sympy.core.expr.Expr method), 101
as_int() (in module sympy.core.compatibility), 169
as_leading_term() (sympy.core.expr.Expr method), 104
as_list() (sympy.polys.polytools.Poly method), 969
as_mutable() (sympy.matrices.expressions.MatrixExpr method), 615
as_mutable() (sympy.matrices.immutable.ImmutableMatrix method), 613
as_mutable() (sympy.matrices.sparse.SparseMatrix method), 602
as_numer_denom() (sympy.core.expr.Expr method), 104
as_ordered_factors() (sympy.core.expr.Expr method), 104
as_ordered_factors() (sympy.core.mul.Mul method), 128
as_ordered_terms() (sympy.core.expr.Expr method), 104
as_poly() (sympy.core.basic.Basic method), 85
as_poly() (sympy.polys.numberfields.AlgebraicNumber method), 1004
as_powers_dict() (sympy.core.expr.Expr method), 104
as_real_imag() (sympy.core.add.Add method), 132
as_real_imag() (sympy.core.expr.Expr method), 104
as_real_imag() (sympy.functions.elementary.complexes.re method), 317
as_real_imag() (sympy.functions.elementary.exponential.exp method), 310
as_real_imag() (sympy.functions.elementary.exponential.log method), 312
as_real_imag() (sympy.functions.elementary.hyperbolic.sinh method), 319
as_relational() (sympy.core.sets.FiniteSet method), 1177
as_relational() (sympy.core.sets.Intersection method), 1178
as_relational() (sympy.core.sets.Interval method), 1175

as_relational() (sympy.core.sets.Union method), 1177
as_sum() (sympy.integrals.Integral method), 529
as_terms() (sympy.core.expr.Expr method), 105
as_two_terms() (sympy.core.add.Add method), 132
as_two_terms() (sympy.core.mul.Mul method), 128
ascents() (sympy.combinatorics.permutations.Permutation method), 182
asec() (in module mpmath), 679
asech() (in module mpmath), 682
asin (class in sympy.functions.elementary.trigonometric), 306
asin() (in module mpmath), 677
asinh (class in sympy.functions.elementary.hyperbolic), 306
asinh() (in module mpmath), 682
ask() (in module sympy.assumptions.ask), 1147
ask_full_inference() (in module sympy.assumptions.ask), 1148
AskAlgebraicHandler (class in sympy.assumptions.handlers.sets), 1154
AskAntiHermitianHandler (class in sympy.assumptions.handlers.sets), 1154
AskBoundedHandler (class in sympy.assumptions.handlers.calculus), 1152
AskComplexHandler (class in sympy.assumptions.handlers.sets), 1154
AskExtendedRealHandler (class in sympy.assumptions.handlers.sets), 1154
AskHermitianHandler (class in sympy.assumptions.handlers.sets), 1154
AskImaginaryHandler (class in sympy.assumptions.handlers.sets), 1155
AskInfinitesimalHandler (class in sympy.assumptions.handlers.calculus), 1153
AskIntegerHandler (class in sympy.assumptions.handlers.sets), 1155
AskNegativeHandler (class in sympy.assumptions.handlers.order), 1154
AskNonZeroHandler (class in sympy.assumptions.handlers.order), 1154
AskOddHandler (class in sympy.assumptions.handlers.ntheory), 1153
AskPositiveHandler (class in sympy.assumptions.handlers.order), 1154
AskPrimeHandler (class in sympy.assumptions.handlers.ntheory), 1153
AskRationalHandler (class in sympy.assumptions.handlers.sets), 1155
AskRealHandler (class in sympy.assumptions.handlers.sets), 1155
assemble_partfrac_list() (in module sympy.polys.partfrac), 1009
AssignmentError, 1128
assoc_laguerre (class in sympy.functions.special.polynomials), 390
assoc_legendre (class in sympy.functions.special.polynomials), 387
assoc_recurrence_memo() (in module sympy.utilities.memoization), 1367
assuming() (in module sympy.assumptions.assume), 1150
assumptions0 (sympy.core.basic.Basic attribute), 86
AssumptionsContext (class in sympy.assumptions.assume), 1149
atan (class in sympy.functions.elementary.trigonometric), 307
atan() (in module mpmath), 678
atan2 (class in sympy.functions.elementary.trigonometric), 307
atan2() (in module mpmath), 679
atanh (class in sympy.functions.elementary.hyperbolic), 307
atanh() (in module mpmath), 682
Atom (class in sympy.core.basic), 97
AtomicExpr (class in sympy.core.expr), 117
atoms() (sympy.combinatorics.permutations.Permutation method), 182
atoms() (sympy.core.basic.Basic method), 86
atoms_table (in module sympy.printing.pretty.pretty_symbology), 1129
auto_number() (in module sympy.parsing.sympy_parser), 1382

auto_symbol() (in module sympy.parsing.sympy_parser), 1382
autoprec() (in module mpmath), 649
autowrap() (in module sympy.utilities.autowrap), 1337

B

B (in module sympy.physics.secondquant), 1407
B (sympy.physics.gaussopt.RayTransferMatrix attribute), 1389
backlunds() (in module mpmath), 847
barnesg() (in module mpmath), 696
base (sympy.combinatorics.perm_groups.PermutationGroup attribute), 202
base (sympy.functions.elementary.exponential.exp attribute), 310
base (sympy.tensor.indexed.Indexed attribute), 1319
base_oneform() (sympy.diffgeom.CoordSystem method), 1582
base_oneforms() (sympy.diffgeom.CoordSystem method), 1582
base_vector() (sympy.diffgeom.CoordSystem method), 1582
base_vectors() (sympy.diffgeom.CoordSystem method), 1582
BaseCovarDerivativeOp (class in sympy.diffgeom), 1585
BasePolynomialError (class in sympy.polys.polyerrors), 1111
BaseScalarField (class in sympy.diffgeom), 1583
BaseSeries (class in sympy.plotting.plot), 1143
baseswap() (sympy.combinatorics.perm_groups.PermutationGroup method), 203
BaseVectorField (class in sympy.diffgeom), 1584
Basic (class in sympy.core.basic), 84
basic_orbits (sympy.combinatorics.perm_groups.PermutationGroup attribute), 204
basic_stabilizers (sympy.combinatorics.perm_groups.PermutationGroup attribute), 204
basic_transversals (sympy.combinatorics.perm_groups.PermutationGroup attribute), 205
Basis() (in module sympy.galgebra.ga), 417
basis() (sympy.polys.agca.modules.FreeModule method), 1015
BBra (in module sympy.physics.secondquant), 1407
Bd (in module sympy.physics.secondquant), 1407
BeamParameter (class in sympy.physics.gaussopt), 1383
bei() (in module mpmath), 736
bell (class in sympy.functions.combinatorial.numbers), 322
bell() (in module mpmath), 865
below() (sympy.printing.pretty.stringpict.stringPict method), 1129
Benini() (in module sympy.stats), 1225
ber() (in module mpmath), 734
berkowitz() (sympy.matrices.matrices.MatrixBase method), 560
berkowitz_charpoly() (sympy.matrices.matrices.MatrixBase method), 561
berkowitz_det() (sympy.matrices.matrices.MatrixBase method), 561
berkowitz_eigenvals() (sympy.matrices.matrices.MatrixBase method), 562
berkowitz_minors() (sympy.matrices.matrices.MatrixBase method), 562
bernfrac() (in module mpmath), 861
bernoulli (class in sympy.functions.combinatorial.numbers), 324
bernoulli() (in module mpmath), 860
Bernoulli() (in module sympy.stats), 1223
bernpoly() (in module mpmath), 862
BesselBase (class in sympy.functions.special.bessel), 361
besseli (class in sympy.functions.special.bessel), 363
besseli() (in module mpmath), 721
besselj (class in sympy.functions.special.bessel), 362
besselj() (in module mpmath), 715
besseljzero() (in module mpmath), 727
besselk (class in sympy.functions.special.bessel), 364
besselk() (in module mpmath), 724
besselsimp() (in module sympy.simplify.simplify), 1192
bessely (class in sympy.functions.special.bessel), 363
bessely() (in module mpmath), 719
besselyzero() (in module mpmath), 729
beta() (in module mpmath), 691
beta() (in module sympy.functions.special.gamma_functions), 338
Beta() (in module sympy.stats), 1226
betainc() (in module mpmath), 692

BetaPrime() (in module sympy.stats), 1227
bihyper() (in module mpmath), 806
bin_to_gray() (sympy.combinatorics.graycode static method), 245
binary_function() (in module sympy.utilities.autowrap), 1338
binary_partitions() (in module sympy.utilities.iterables), 1348
binomial (class in sympy.functions.combinatorial.factorials), 325
binomial() (in module mpmath), 685
Binomial() (in module sympy.stats), 1224
binomial_coefficients() (in module sympy.ntheory.multinomial), 277
binomial_coefficients_list() (in module sympy.ntheory.multinomial), 278
Bisection (class in mpmath.calculus.optimization), 880
bisectors() (sympy.geometry.polygon.Triangle method), 492
bitlist_from_subset() (sympy.combinatorics.subsets.Subset class method), 235
BKet (in module sympy.physics.secondquant), 1407
block_collapse() (in module sympy.matrices.expressions.blockmatrix), 619
BlockDiagMatrix (class in sympy.matrices.expressions.blockmatrix), 619
BlockMatrix (class in sympy.matrices.expressions.blockmatrix), 618
bm (sympy.functions.special.hyper.meijerg attribute), 378
BosonicBasis (class in sympy.physics.secondquant), 1408
bother (sympy.functions.special.hyper.meijerg attribute), 378
bq (sympy.functions.special.hyper.hyper attribute), 375
bq (sympy.functions.special.hyper.meijerg attribute), 378
Bra (class in sympy.physics.quantum.state), 1543
bra (sympy.physics.quantum.operator.OuterProduct attribute), 1523
bra (sympy.physics.secondquant.InnerProduct attribute), 1408
BraBase (class in sympy.physics.quantum.state), 1542
bracelets() (in module sympy.utilities.iterables), 1348
bsgs_direct_product() (in module sympy.combinatorics.tensor_can), 260
bspline_basis() (in module sympy.functions.special.bsplines), 367
bspline_basis_set() (in module sympy.functions.special.bsplines), 368
build_expression_tree() (in module sympy.series.gruntz), 1167

C

C (in module sympy.core.basic), 97
C (sympy.matrices.immutable.ImmutableMatrix attribute), 613
C (sympy.matrices.matrices.MatrixBase attribute), 556
C (sympy.physics.gaussopt.RayTransferMatrix attribute), 1389
cacheit() (in module sympy.core.cache), 84
calc_nodes() (mpmath.calculus.quadrature.GaussLegendre method), 909
calc_nodes() (mpmath.calculus.quadrature.QuadratureRule method), 906
calc_nodes() (mpmath.calculus.quadrature.TanhSinh method), 908
calculate_series() (in module sympy.series.gruntz), 1167
cancel() (in module sympy.polys.polytools), 964
cancel() (sympy.core.expr.Expr method), 105
cancel() (sympy.polys.polyclasses.DMF method), 1049
cancel() (sympy.polys.polyclasses.DMP method), 1044
cancel() (sympy.polys.polytools.Poly method), 969
canon_args() (sympy.tensor.tensor.TensMul method), 1332
canon_bp() (in module sympy.tensor.tensor), 1335
canon_bp() (sympy.tensor.tensor.TensAdd method), 1330
canon_bp() (sympy.tensor.tensor.TensMul method), 1332
canonical_variables (sympy.core.basic.Basic attribute), 87
canonicalize() (in module sympy.combinatorics.tensor_can), 255

capture() (in module sympy.utilities.iterables), 1348
cardinality (sympy.combinatorics.permutations.Permutation attribute), 182
cardinality (sympy.combinatorics.subsets.Subset attribute), 235
casoratian() (in module sympy.matrices.dense), 591
catalan (class in sympy.functions.combinatorial.numbers), 326
catalan (mpmath.mp attribute), 657
Category (class in sympy.categories), 1567
Cauchy() (in module sympy.stats), 1228
cbrt() (in module mpmath), 658
ccode() (in module sympy.printing.ccode), 1117
CCodeGen (class in sympy.utilities.codegen), 1342
CCodePrinter (class in sympy.printing.ccode), 1117
cdf() (sympy.statistics.distributions.Normal method), 1219
cdf() (sympy.statistics.distributions.PDF method), 1221
cdf() (sympy.statistics.distributions.Uniform method), 1220
ceil() (in module mpmath), 643
ceiling (class in sympy.functions.elementary.integers), 307
center (sympy.geometry.ellipse.Ellipse attribute), 467
center (sympy.geometry.polygon.RegularPolygon attribute), 486
center() (sympy.combinatorics.perm_groups.PermutationGroup method), 205
central_inertia (sympy.physics.mechanics.rigidbody.RigidBody attribute), 1496
centralizer() (sympy.combinatorics.perm_groups.PermutationGroup method), 206
centroid (sympy.geometry.polygon.Polygon attribute), 480
centroid (sympy.geometry.polygon.RegularPolygon attribute), 486
centroid() (in module sympy.geometry.util), 440
CG (class in sympy.physics.quantum.cg), 1511
cg_simp() (in module sympy.physics.quantum.cg), 1512
CGate (class in sympy.physics.quantum.gate), 1551
characteristic() (sympy.polys.domains.domain.Domain method), 1032
characteristic() (sympy.polys.domains.FiniteField method), 1037
charpoly() (sympy.matrices.matrices.MatrixBase method), 562
chebyfit() (in module mpmath), 912
chebyshevt (class in sympy.functions.special.polynomials), 384
chebyshevt_poly() (in module sympy.polys.orthopolys), 1006
chebyshevt_root (class in sympy.functions.special.polynomials), 386
chebyshevu (class in sympy.functions.special.polynomials), 385
chebyshevu_poly() (in module sympy.polys.orthopolys), 1006
chebyshevu_root (class in sympy.functions.special.polynomials), 386
chebyt() (in module mpmath), 778
chebyu() (in module mpmath), 779
check_assumptions() (in module sympy.solvers.solvers), 1303
check_output() (sympy.utilities.runtests.SymPyOutputCheck method), 1372
checkinfsol() (in module sympy.solvers.ode), 1263
checkodesol() (in module sympy.solvers.ode), 1260
checkpdesol() (in module sympy.solvers.pde), 1291
checksol() (in module sympy.solvers.solvers), 1304
Chi (class in sympy.functions.special.error_functions), 350
chi() (in module mpmath), 709
Chi() (in module sympy.stats), 1228
ChiNoncentral() (in module sympy.stats), 1229
ChiSquared() (in module sympy.stats), 1229
cholesky() (in module mpmath), 920
cholesky() (sympy.matrices.matrices.MatrixBase method), 562
cholesky() (sympy.matrices.sparse.SparseMatrix method), 602
cholesky_solve() (sympy.matrices.matrices.MatrixBase method), 563
chop() (in module mpmath), 644
Ci (class in sympy.functions.special.error_functions), 348
ci() (in module mpmath), 707
Circle (class in sympy.geometry.ellipse), 474

circuit_plot() (in module sympy.physics.quantum.circuitplot), 1550
CircuitPlot (class in sympy.physics.quantum.circuitplot), 1549
circumcenter (sympy.geometry.polygon.RegularPolygon attribute), 486
circumcenter (sympy.geometry.polygon.Triangle attribute), 493
circumcircle (sympy.geometry.polygon.RegularPolygon attribute), 487
circumcircle (sympy.geometry.polygon.Triangle attribute), 493
circumference (sympy.geometry.ellipse.Circle attribute), 475
circumference (sympy.geometry.ellipse.Ellipse attribute), 467
circumradius (sympy.geometry.polygon.RegularPolygon attribute), 487
circumradius (sympy.geometry.polygon.Triangle attribute), 493
CL (sympy.matrices.sparse.SparseMatrix attribute), 601
class_key() (sympy.core.add.Add class method), 132
class_key() (sympy.core.basic.Basic class method), 87
classify_ode() (in module sympy.solvers.ode), 1259
classify_pde() (in module sympy.solvers.pde), 1290
classof() (in module sympy.matrices.matrices), 588
clcos() (in module mpmath), 853
clear() (mpmath.calculus.quadrature.QuadratureRule method), 906
clear() (sympy.tensor.tensor._TensorManager method), 1325
clear_denoms() (sympy.polys.polyclasses.DMP method), 1044
clear_denoms() (sympy.polys.polytools.Poly method), 970
clebsch_gordan() (in module sympy.physics.wigner), 1418
clsin() (in module mpmath), 851
CMod (class in sympy.physics.quantum.shor), 1562
CNOT (in module sympy.physics.quantum.gate), 1553
CNotGate (class in sympy.physics.quantum.gate), 1553
CodeGen (class in sympy.utilities.codegen), 1342
codegen() (in module sympy.utilities.codegen), 1344
CodePrinter (class in sympy.printing.codeprinter), 1127
CodeWrapper (class in sympy.utilities.autowrap), 1337
codomain (sympy.categories.CompositeMorphism attribute), 1565
codomain (sympy.categories.Morphism attribute), 1563
coeff() (sympy.core.expr.Expr method), 105
coeff_monomial() (sympy.polys.polytools.Poly method), 970
coefficients (sympy.geometry.line.LinearEntity attribute), 448
coeffs() (sympy.polys.numberfields.AlgebraicNumber method), 1004
coeffs() (sympy.polys.polyclasses.DMP method), 1044
coeffs() (sympy.polys.polytools.Poly method), 971
CoercionFailed (class in sympy.polys.polyerrors), 1112
cofactor() (sympy.matrices.matrices.MatrixBase method), 563
cofactorMatrix() (sympy.matrices.matrices.MatrixBase method), 563
cofactors() (in module sympy.polys.polytools), 956
cofactors() (sympy.core.numbers.Number method), 121
cofactors() (sympy.polys.domains.domain.Domain method), 1032
cofactors() (sympy.polys.polyclasses.DMP method), 1044
cofactors() (sympy.polys.polytools.Poly method), 971
Coin() (in module sympy.stats), 1224
col() (sympy.matrices.sparse.SparseMatrix method), 603
col_del() (sympy.matrices.dense.MutableDenseMatrix method), 595
col_del() (sympy.matrices.sparse.MutableSparseMatrix method), 607
col_insert() (sympy.matrices.matrices.MatrixBase method), 563
col_join() (sympy.matrices.matrices.MatrixBase method), 563
col_join() (sympy.matrices.sparse.MutableSparseMatrix method), 608
col_list() (sympy.matrices.sparse.SparseMatrix method), 603
col_op() (sympy.matrices.dense.MutableDenseMatrix method), 596

col_op() (sympy.matrices.sparse.MutableSparseMatrix method), 608
col_swap() (sympy.matrices.dense.MutableDenseMatrix method), 596
col_swap() (sympy.matrices.sparse.MutableSparseMatrix method), 609
collect() (in module sympy.simplify.simplify), 1184
collect() (sympy.core.expr.Expr method), 107
collect_const() (in module sympy.simplify.simplify), 1196
collect_sqrt() (in module sympy.simplify.simplify), 1196
combsimp() (in module sympy.simplify.simplify), 1193
combsimp() (sympy.core.expr.Expr method), 107
comm_i2symbol() (sympy.tensor.tensor._TensorManager method), 1325
comm_symbols2i() (sympy.tensor.tensor._TensorManager method), 1325
common_prefix() (in module sympy.utilities.iterables), 1348
common_suffix() (in module sympy.utilities.iterables), 1348
commutative_diagrams (sympy.categories.Category attribute), 1567
Commutator (class in sympy.physics.quantum.commutator), 1513
Commutator (class in sympy.physics.secondquant), 1409
commutator() (sympy.combinatorics.perm_groups.PermutationGroup method), 206
commutator() (sympy.combinatorics.permutations.Permutation method), 182
commutes_with() (sympy.combinatorics.permutations.Permutation method), 183
commutes_with() (sympy.tensor.tensor.TensorHead method), 1329
comp() (in module sympy.utilities.randtest), 1370
compare() (in module sympy.series.gruntz), 1167
compare() (sympy.core.basic.Basic method), 87
compare_pretty() (sympy.core.basic.Basic static method), 88
complement (sympy.core.sets.Set attribute), 1173
ComplexSpace (class in sympy.physics.quantum.hilbert), 1519
components (sympy.categories.CompositeMorphism attribute), 1565
compose() (in module sympy.polys.polytools), 959
compose() (sympy.categories.Morphism method), 1563
compose() (sympy.polys.polyclasses.DMP method), 1044
compose() (sympy.polys.polytools.Poly method), 971
CompositeDomain (class in sympy.polys.domains.compositedomain), 1037
CompositeMorphism (class in sympy.categories), 1565
ComputationFailed (class in sympy.polys.polyerrors), 1112
compute_known_facts() (in module sympy.assumptions.ask), 1148
compute_leading_term() (sympy.core.expr.Expr method), 107
conclusions (sympy.categories.Diagram attribute), 1569
cond (sympy.functions.elementary.piecewise.ExprCondPair attribute), 310
condition_number() (sympy.matrices.matrices.MatrixBase method), 564
ConditionalDomain (class in sympy.stats.rv), 1255
confidence() (sympy.statistics.distributions.Normal method), 1219
confidence() (sympy.statistics.distributions.Uniform method), 1220
conjugate() (in module mpmath), 642
conjugate (class in sympy.functions.elementary.complexes), 308
conjugate (sympy.combinatorics.partitions.IntegerPartition attribute), 174
conjugate_gauss_beams() (in module sympy.physics.gaussopt), 1390
connect_to() (sympy.diffgeom.CoordSystem method), 1582
conserve_mpmath_dps() (in module sympy.utilities.decorator), 1345
constant_renumber() (in module sympy.solvers.ode), 1265
constantsimp() (in module sympy.solvers.ode), 1265
construct_domain() (in module sympy.polys.constructor), 1002

contains() (sympy.combinatorics.perm_groups.PermutationGroup method), 207
contains() (sympy.core.sets.Set method), 1173
contains() (sympy.geometry.line.Line method), 455
contains() (sympy.geometry.line.LinearEntity method), 448
contains() (sympy.geometry.line.Ray method), 457
contains() (sympy.geometry.line.Segment method), 459
contains() (sympy.polys.agca.ideals.Ideal method), 1021
contains() (sympy.polys.agca.modules.Module method), 1014
contains() (sympy.polys.polytools.GroebnerBasis method), 999
contains() (sympy.series.order.Order method), 1170
content() (in module sympy.polys.polytools), 958
content() (sympy.polys.polyclasses.DMP method), 1044
content() (sympy.polys.polytools.Poly method), 971
continued_fraction() (in module sympy.physics.quantum.shor), 1562
ContinuousDomain (class in sympy.stats.crv), 1255
ContinuousProbability (class in sympy.statistics.distributions), 1218
ContinuousPSpace (class in sympy.stats.crv), 1255
ContinuousRV() (in module sympy.stats), 1250
contract_metric() (sympy.tensor.tensor.TensAdd method), 1330
contract_metric() (sympy.tensor.tensor.TensMul method), 1332
contraction() (in module sympy.physics.secondquant), 1410
control_line() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1549
control_point() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1550
controls (sympy.physics.quantum.gate.CGate attribute), 1551
controls (sympy.physics.quantum.gate.CNotGate attribute), 1553
convergence_statement (sympy.functions.special.hyper.hyper attribute), 375
convert() (sympy.polys.agca.modules.FreeModule method), 1015
convert() (sympy.polys.agca.modules.Module method), 1015
convert() (sympy.polys.agca.modules.QuotientModule method), 1023
convert() (sympy.polys.agca.modules.SubModule method), 1017
convert() (sympy.polys.domains.domain.Domain method), 1032
convert() (sympy.polys.polyclasses.DMP method), 1044
convert_from() (sympy.polys.domains.domain.Domain method), 1032
convert_to_native_paths() (in module sympy.utilities.runtests), 1372
convert_xor() (in module sympy.parsing.sympy_parser), 1382
convex_hull() (in module sympy.geometry.util), 438
coord_function() (sympy.diffgeom.CoordSystem method), 1582
coord_functions() (sympy.diffgeom.CoordSystem method), 1582
coord_tuple_transform_to() (sympy.diffgeom.CoordSystem method), 1582
CoordSystem (class in sympy.diffgeom), 1580
copyin_list() (sympy.matrices.dense.MutableDenseMatrix method), 596
copyin_matrix() (sympy.matrices.dense.MutableDenseMatrix method), 597
corners (sympy.combinatorics.polyhedron.Polyhedron attribute), 229
cos (class in sympy.functions.elementary.trigonometric), 308
cos() (in module mpmath), 673
coset_factor() (sympy.combinatorics.perm_groups.PermutationGroup method), 208
coset_rank() (sympy.combinatorics.perm_groups.PermutationGroup method), 208
coset_unrank() (sympy.combinatorics.perm_groups.PermutationGroup method), 209
cosh (class in sympy.functions.elementary.hyperbolic), 369
cosh() (in module mpmath), 680
cosine_transform() (in module sympy.integrals.transforms), 503
cosm() (in module mpmath), 923
cospi() (in module mpmath), 676
cot (class in sympy.functions.elementary.trigonometric), 309
cot() (in module mpmath), 676

coth (class in sympy.functions.elementary.hyperbolic), 310
coth() (in module mpmath), 682
could_extract_minus_sign() (sympy.core.expr.Expr method), 107
coulombc() (in module mpmath), 767
coulombf() (in module mpmath), 761
coulombg() (in module mpmath), 765
count() (sympy.core.basic.Basic method), 88
count_complex_roots() (sympy.polys.polyclasses.DMP method), 1044
count_ops() (in module sympy.core.function), 162
count_ops() (sympy.core.basic.Basic method), 88
count_ops() (sympy.core.expr.Expr method), 107
count_real_roots() (sympy.polys.polyclasses.DMP method), 1044
count_roots() (in module sympy.polys.polytools), 962
count_roots() (sympy.polys.polytools.Poly method), 972
couple() (in module sympy.physics.quantum.spin), 1540
CovarDerivativeOp (class in sympy.diffgeom), 1586
cplot() (in module mpmath), 654
CreateBoson (class in sympy.physics.secondquant), 1403
CreateFermion (class in sympy.physics.secondquant), 1405
cross() (in module sympy.physics.mechanics.functions), 1485
cross() (sympy.matrices.matrices.MatrixBase method), 564
cross() (sympy.physics.mechanics.essential.Dyadic method), 1483
cross() (sympy.physics.mechanics.essential.Vector method), 1480
crt() (in module sympy.ntheory.modular), 275
crt1() (in module sympy.ntheory.modular), 276
crt2() (in module sympy.ntheory.modular), 276
csc() (in module mpmath), 675
csch() (in module mpmath), 682
cse() (in module sympy.simplify.cse_main), 1200
current (sympy.combinatorics.graycode.GrayCode attribute), 242
Curve (class in sympy.geometry.curve), 461
CurvedMirror (class in sympy.physics.gaussopt), 1385
CurvedRefraction (class in sympy.physics.gaussopt), 1385
Cycle (class in sympy.combinatorics.permutations), 198
cycle_length() (in module sympy.ntheory.generate), 266
cycle_structure (sympy.combinatorics.permutations.Permuta attribute), 183
cycles (sympy.combinatorics.permutations.Permutation attribute), 184
cyclic() (sympy.combinatorics.generators static method), 200
cyclic_form (sympy.combinatorics.permutations.Permutatio attribute), 184
cyclic_form (sympy.combinatorics.polyhedron.Polyhedron attribute), 229
CyclicGroup() (in module sympy.combinatorics.named_groups), 246
cyclotomic() (in module mpmath), 870
cyclotomic_poly() (in module sympy.polys.specialpolys), 1005
CythonCodeWrapper (class in sympy.utilities.autowrap), 1337

D

D (sympy.matrices.matrices.MatrixBase attribute), 556
D (sympy.physics.gaussopt.RayTransferMatrix attribute), 1389
D() (sympy.physics.quantum.spin.Rotation class method), 1532
d() (sympy.physics.quantum.spin.Rotation class method), 1533
Dagger (class in sympy.physics.quantum.dagger), 1514
Dagger (class in sympy.physics.secondquant), 1398
Dagum() (in module sympy.stats), 1230
DataType (class in sympy.utilities.codegen), 1341
dcm() (sympy.physics.mechanics.essential.ReferenceFrame method), 1476
DD() (in module sympy.galgebra.ga), 417
debug() (in module sympy.utilities.misc), 1367
decompose() (in module sympy.polys.polytools), 959
decompose() (sympy.physics.quantum.gate.CGate method), 1551

decompose() (sympy.physics.quantum.gate.SwapGate method), 1553
decompose() (sympy.physics.quantum.qft.IQFT method), 1557
decompose() (sympy.physics.quantum.qft.QFT method), 1557
decompose() (sympy.polys.polyclasses.DMP method), 1044
decompose() (sympy.polys.polytools.Poly method), 972
deflate() (sympy.polys.polyclasses.DMP method), 1044
deflate() (sympy.polys.polytools.Poly method), 972
degree (mpmath.mp attribute), 657
degree (sympy.combinatorics.perm_groups.PermutationGroup attribute), 209
degree() (in module sympy.polys.polytools), 951
degree() (sympy.polys.polyclasses.DMP method), 1044
degree() (sympy.polys.polytools.Poly method), 972
degree_list() (in module sympy.polys.polytools), 951
degree_list() (sympy.polys.polyclasses.DMP method), 1044
degree_list() (sympy.polys.polytools.Poly method), 972
degrees() (in module mpmath), 673
delta (sympy.functions.special.hyper.meijerg attribute), 378
deltaintegrate() (sympy.integrals static method), 526
denom() (sympy.polys.domains.AlgebraicField method), 1040
denom() (sympy.polys.domains.domain.Domain method), 1032
denom() (sympy.polys.domains.ExpressionDomain method), 1042
denom() (sympy.polys.domains.FractionField method), 1041
denom() (sympy.polys.domains.ring.Ring method), 1035
denom() (sympy.polys.polyclasses.DMF method), 1049
density() (in module sympy.stats), 1252
depth() (sympy.polys.agca.ideals.Ideal method), 1021
Derivative (class in sympy.core.function), 150
derived_series() (sympy.combinatorics.perm_groups.PermutationGroup method), 209
derived_subgroup() (sympy.combinatorics.perm_groups.PermutationGroup method), 210
descents() (sympy.combinatorics.permutations.Permutation method), 184
det() (sympy.matrices.matrices.MatrixBase method), 564
det_bareis() (sympy.matrices.matrices.MatrixBase method), 564
det_LU_decomposition() (sympy.matrices.matrices.MatrixBase method), 564
diag() (in module sympy.matrices.dense), 589
diagonal_solve() (sympy.matrices.matrices.MatrixBase method), 565
diagonalize() (sympy.matrices.matrices.MatrixBase method), 565
Diagram (class in sympy.categories), 1568
DiagramGrid (class in sympy.categories.diagram_drawing), 1571
Dict (class in sympy.core.containers), 168
dict_merge() (in module sympy.utilities.iterables), 1349
Die() (in module sympy.stats), 1223
DiePSpace (class in sympy.stats.frv_types), 1256
diff() (in module mpmath), 896
diff() (in module sympy.core.function), 153
diff() (sympy.matrices.matrices.MatrixBase method), 566
diff() (sympy.physics.mechanics.essential.Vector method), 1480
diff() (sympy.polys.polyclasses.DMP method), 1045
diff() (sympy.polys.polytools.Poly method), 973
Differential (class in sympy.diffgeom), 1585
DifferentialOperator (class in sympy.physics.quantum.operator), 1524
differint() (in module mpmath), 899
diffs() (in module mpmath), 897
diffs_exp() (in module mpmath), 898
diffs_prod() (in module mpmath), 898
digamma() (in module mpmath), 699
digamma() (in module sympy.functions.special.gamma_functions), 339
digit_2txt (in module sympy.printing.pretty.pretty_symbology), 1128
dihedral() (sympy.combinatorics.generators static method), 200
DihedralGroup() (in module sympy.combinatorics.named_groups), 246

dimension (sympy.physics.quantum.hilbert.HilbertSpace attribute), 1519
DiracDelta (class in sympy.functions.special.delta_functions), 336
DirectProduct() (in module sympy.combinatorics.group_constructs), 254
dirichlet() (in module mpmath), 841
dirichlet_eta (class in sympy.functions.special.zeta_functions), 370
DiscreteUniform() (in module sympy.stats), 1223
discriminant() (in module sympy.polys.polytools), 955
discriminant() (sympy.polys.polyclasses.DMP method), 1045
discriminant() (sympy.polys.polytools.Poly method), 973
distance() (sympy.geometry.line.Segment method), 459
distance() (sympy.geometry.point.Point method), 441
distance() (sympy.geometry.polygon.Polygon method), 480
div() (in module sympy.polys.polytools), 953
div() (sympy.polys.domains.domain.Domain method), 1032
div() (sympy.polys.domains.field.Field method), 1034
div() (sympy.polys.domains.ring.Ring method), 1035
div() (sympy.polys.polyclasses.DMP method), 1045
div() (sympy.polys.polytools.Poly method), 973
divergence (sympy.physics.gaussopt.BeamParameter attribute), 1383
divisor_count() (in module sympy.ntheory.factor_), 275
divisors() (in module sympy.ntheory.factor_), 274
DMF (class in sympy.polys.polyclasses), 1049
DMP (class in sympy.polys.polyclasses), 1044
dmp_abs() (in module sympy.polys.densearith), 1065
dmp_add() (in module sympy.polys.densearith), 1065
dmp_add_ground() (in module sympy.polys.densearith), 1063
dmp_add_mul() (in module sympy.polys.densearith), 1065
dmp_add_term() (in module sympy.polys.densearith), 1062
dmp_apply_pairs() (in module sympy.polys.densebasic), 1062
dmp_cancel() (in module sympy.polys.euclidtools), 1105
dmp_clear_denoms() (in module sympy.polys.densetools), 1078
dmp_compose() (in module sympy.polys.densetools), 1077
dmp_content() (in module sympy.polys.euclidtools), 1105
dmp_convert() (in module sympy.polys.densebasic), 1054
dmp_copy() (in module sympy.polys.densebasic), 1053
dmp_deflate() (in module sympy.polys.densebasic), 1059
dmp_degree() (in module sympy.polys.densebasic), 1052
dmp_degree_in() (in module sympy.polys.densebasic), 1052
dmp_degree_list() (in module sympy.polys.densebasic), 1052
dmp_diff() (in module sympy.polys.densetools), 1071
dmp_diff_eval_in() (in module sympy.polys.densetools), 1072
dmp_diff_in() (in module sympy.polys.densetools), 1071
dmp_discriminant() (in module sympy.polys.euclidtools), 1102
dmp_div() (in module sympy.polys.densearith), 1068
dmp_eject() (in module sympy.polys.densebasic), 1061
dmp_euclidean_prs() (in module sympy.polys.euclidtools), 1099
dmp_eval() (in module sympy.polys.densetools), 1071
dmp_eval_in() (in module sympy.polys.densetools), 1071
dmp_eval_tail() (in module sympy.polys.densetools), 1072
dmp_exclude() (in module sympy.polys.densebasic), 1060
dmp_expand() (in module sympy.polys.densearith), 1070
dmp_exquo() (in module sympy.polys.densearith), 1069
dmp_exquo_ground() (in module sympy.polys.densearith), 1064
dmp_ext_factor() (in module sympy.polys.factortools), 1109

dmp_factor_list() (in module sympy.polys.factortools), 1109
dmp_factor_list_include() (in module sympy.polys.factortools), 1109
dmp_ff_div() (in module sympy.polys.densearith), 1068
dmp_ff_prs_gcd() (in module sympy.polys.euclidtools), 1102
dmp_from_dict() (in module sympy.polys.densebasic), 1058
dmp_from_sympy() (in module sympy.polys.densebasic), 1054
dmp_gcd() (in module sympy.polys.euclidtools), 1104
dmp_gcdex() (in module sympy.polys.euclidtools), 1099
dmp_ground() (in module sympy.polys.densebasic), 1056
dmp_ground_content() (in module sympy.polys.densetools), 1074
dmp_ground_extract() (in module sympy.polys.densetools), 1075
dmp_ground_LC() (in module sympy.polys.densebasic), 1051
dmp_ground_monic() (in module sympy.polys.densetools), 1073
dmp_ground_nth() (in module sympy.polys.densebasic), 1055
dmp_ground_p() (in module sympy.polys.densebasic), 1056
dmp_ground_primitive() (in module sympy.polys.densetools), 1075
dmp_ground_TC() (in module sympy.polys.densebasic), 1051
dmp_ground_trunc() (in module sympy.polys.densetools), 1073
dmp_grounds() (in module sympy.polys.densebasic), 1057
dmp_half_gcdex() (in module sympy.polys.euclidtools), 1099
dmp_include() (in module sympy.polys.densebasic), 1060
dmp_inflate() (in module sympy.polys.densebasic), 1060
dmp_inject() (in module sympy.polys.densebasic), 1060
dmp_inner_gcd() (in module sympy.polys.euclidtools), 1104
dmp_inner_subresultants() (in module sympy.polys.euclidtools), 1100
dmp_integrate() (in module sympy.polys.densetools), 1070
dmp_integrate_in() (in module sympy.polys.densetools), 1070
dmp_invert() (in module sympy.polys.euclidtools), 1099
dmp_irreducible_p() (in module sympy.polys.factortools), 1109
dmp_l1_norm() (in module sympy.polys.densearith), 1070
dmp_LC() (in module sympy.polys.densebasic), 1050
dmp_lcm() (in module sympy.polys.euclidtools), 1104
dmp_lift() (in module sympy.polys.densetools), 1077
dmp_list_terms() (in module sympy.polys.densebasic), 1061
dmp_max_norm() (in module sympy.polys.densearith), 1069
dmp_mul() (in module sympy.polys.densearith), 1066
dmp_mul_ground() (in module sympy.polys.densearith), 1063
dmp_mul_term() (in module sympy.polys.densearith), 1063
dmp_multi_deflate() (in module sympy.polys.densebasic), 1059
dmp_neg() (in module sympy.polys.densearith), 1065
dmp_negative_p() (in module sympy.polys.densebasic), 1057
dmp_nest() (in module sympy.polys.densebasic), 1059
dmp_normal() (in module sympy.polys.densebasic), 1054
dmp_nth() (in module sympy.polys.densebasic), 1055
dmp_one() (in module sympy.polys.densebasic), 1056
dmp_one_p() (in module sympy.polys.densebasic), 1056
dmp_pdiv() (in module sympy.polys.densearith), 1067
dmp_permute() (in module sympy.polys.densebasic), 1058
dmp_pexquo() (in module sympy.polys.densearith), 1067
dmp_positive_p() (in module sympy.polys.densebasic), 1057
dmp_pow() (in module sympy.polys.densearith), 1066
dmp_pquo() (in module sympy.polys.densearith), 1067
dmp_prem() (in module sympy.polys.densearith), 1067
dmp_primitive() (in module sympy.polys.euclidtools), 1105

dmp_primitive_prs() (in module sympy.polys.euclidtools), 1099
dmp_prs_resultant() (in module sympy.polys.euclidtools), 1100
dmp_qq_collins_resultant() (in module sympy.polys.euclidtools), 1101
dmp_qq_heu_gcd() (in module sympy.polys.euclidtools), 1103
dmp_quo() (in module sympy.polys.densearith), 1069
dmp_quo_ground() (in module sympy.polys.densearith), 1064
dmp_raise() (in module sympy.polys.densebasic), 1059
dmp_rem() (in module sympy.polys.densearith), 1068
dmp_resultant() (in module sympy.polys.euclidtools), 1102
dmp_revert() (in module sympy.polys.densetools), 1078
dmp_rr_div() (in module sympy.polys.densearith), 1068
dmp_rr_prs_gcd() (in module sympy.polys.euclidtools), 1102
dmp_slice() (in module sympy.polys.densebasic), 1062
dmp_sqr() (in module sympy.polys.densearith), 1066
dmp_strip() (in module sympy.polys.densebasic), 1052
dmp_sub() (in module sympy.polys.densearith), 1065
dmp_sub_ground() (in module sympy.polys.densearith), 1063
dmp_sub_mul() (in module sympy.polys.densearith), 1066
dmp_sub_term() (in module sympy.polys.densearith), 1062
dmp_subresultants() (in module sympy.polys.euclidtools), 1100
dmp_swap() (in module sympy.polys.densebasic), 1058
dmp_TC() (in module sympy.polys.densebasic), 1051
dmp_terms_gcd() (in module sympy.polys.densebasic), 1061
dmp_to_dict() (in module sympy.polys.densebasic), 1058
dmp_to_tuple() (in module sympy.polys.densebasic), 1053
dmp_trial_division() (in module sympy.polys.factortools), 1105
dmp_true_LT() (in module sympy.polys.densebasic), 1051
dmp_trunc() (in module sympy.polys.densetools), 1072
dmp_validate() (in module sympy.polys.densebasic), 1053
dmp_zero() (in module sympy.polys.densebasic), 1055
dmp_zero_p() (in module sympy.polys.densebasic), 1055
dmp_zeros() (in module sympy.polys.densebasic), 1057
dmp_zz_collins_resultant() (in module sympy.polys.euclidtools), 1101
dmp_zz_diophantine() (in module sympy.polys.factortools), 1108
dmp_zz_factor() (in module sympy.polys.factortools), 1108
dmp_zz_heu_gcd() (in module sympy.polys.euclidtools), 1103
dmp_zz_mignotte_bound() (in module sympy.polys.factortools), 1105
dmp_zz_modular_resultant() (in module sympy.polys.euclidtools), 1101
dmp_zz_wang() (in module sympy.polys.factortools), 1108
dmp_zz_wang_hensel_lifting() (in module sympy.polys.factortools), 1108
dmp_zz_wang_lead_coeffs() (in module sympy.polys.factortools), 1108
dmp_zz_wang_non_divisors() (in module sympy.polys.factortools), 1108
dmp_zz_wang_test_points() (in module sympy.polys.factortools), 1108
doctest() (in module sympy.utilities.runtests), 1372
doctest_depends_on() (in module sympy.utilities.decorator), 1345
doit() (sympy.core.basic.Basic method), 88
doit() (sympy.functions.elementary.piecewise.Piecewise method), 316
doit() (sympy.integrals.Integral method), 531
doit() (sympy.physics.mechanics.essential.Dyadic method), 1483
doit() (sympy.physics.mechanics.essential.Vector method), 1480
doit() (sympy.physics.quantum.anticommutator.AntiCommut method), 1511
doit() (sympy.physics.quantum.commutator.Commutator method), 1514
doit() (sympy.physics.secondquant.AntiSymmetricTensor method), 1415
doit() (sympy.physics.secondquant.Commutator method), 1410
doit() (sympy.physics.secondquant.NO method), 1412

doit() (sympy.series.limits.Limit method), 1165
doit_numerically() (sympy.core.function.Derivative method), 152
Domain (class in sympy.polys.domains.domain), 1031
domain (sympy.categories.CompositeMorphism attribute), 1566
domain (sympy.categories.Morphism attribute), 1564
domain (sympy.polys.polytools.Poly attribute), 973
DomainError (class in sympy.polys.polyerrors), 1112
doprint() (sympy.printing.ccode.CCodePrinter method), 1117
doprint() (sympy.printing.fcode.FCodePrinter method), 1119
doprint() (sympy.printing.mathml.MathMLPrinter method), 1123
doprint() (sympy.printing.printer.Printer method), 1115
dot() (in module sympy.physics.mechanics.functions), 1485
dot() (sympy.geometry.point.Point method), 442
dot() (sympy.matrices.matrices.MatrixBase method), 566
dot() (sympy.physics.mechanics.essential.Dyadic method), 1483
dot() (sympy.physics.mechanics.essential.Vector method), 1481
dotprint() (in module sympy.printing.dot), 1131
double_coset_can_rep() (in module sympy.combinatorics.tensor_can), 257
draw() (sympy.categories.diagram_drawing.XypicDiagramDrawer method), 1578
dsolve() (in module sympy.solvers.ode), 1257
dt() (sympy.physics.mechanics.essential.Dyadic method), 1483
dt() (sympy.physics.mechanics.essential.Vector method), 1481
dtype (sympy.polys.agca.modules.FreeModule attribute), 1016
dtype (sympy.polys.agca.modules.QuotientModule attribute), 1023
dtype (sympy.polys.domains.AlgebraicField attribute), 1040
dtype (sympy.polys.domains.ExpressionDomain attribute), 1042
dtype (sympy.polys.domains.FractionField attribute), 1041
dual (sympy.physics.quantum.state.StateBase attribute), 1542
dual() (sympy.matrices.matrices.MatrixBase method), 566
dual_class() (sympy.physics.quantum.state.StateBase class method), 1542
Dummy (class in sympy.core.symbol), 118
dummy_eq() (sympy.core.basic.Basic method), 89
DummyWrapper (class in sympy.utilities.autowrap), 1337
dump_c() (sympy.utilities.codegen.CCodeGen method), 1342
dump_code() (sympy.utilities.codegen.CodeGen method), 1342
dump_f95() (sympy.utilities.codegen.FCodeGen method), 1343
dump_h() (sympy.utilities.codegen.CCodeGen method), 1343
dump_h() (sympy.utilities.codegen.FCodeGen method), 1344
dump_pyx() (sympy.utilities.autowrap.CythonCodeWrapper method), 1337
dup_content() (in module sympy.polys.densetools), 1074
dup_cyclotomic_p() (in module sympy.polys.factortools), 1106
dup_decompose() (in module sympy.polys.densetools), 1077
dup_extract() (in module sympy.polys.densetools), 1075
dup_gf_factor() (in module sympy.polys.factortools), 1109
dup_lshift() (in module sympy.polys.densearith), 1064
dup_mirror() (in module sympy.polys.densetools), 1076
dup_monic() (in module sympy.polys.densetools), 1073
dup_primitive() (in module sympy.polys.densetools), 1074
dup_random() (in module sympy.polys.densebasic), 1062
dup_real_imag() (in module sympy.polys.densetools), 1076
dup_reverse() (in module sympy.polys.densebasic), 1053
dup_rshift() (in module sympy.polys.densearith), 1064
dup_scale() (in module sympy.polys.densetools), 1076
dup_shift() (in module sympy.polys.densetools), 1076

dup_sign_variations() (in module sympy.polys.densetools), 1078
dup_transform() (in module sympy.polys.densetools), 1076
dup_zz_cyclotomic_factor() (in module sympy.polys.factortools), 1107
dup_zz_cyclotomic_poly() (in module sympy.polys.factortools), 1107
dup_zz_factor() (in module sympy.polys.factortools), 1107
dup_zz_factor_sqf() (in module sympy.polys.factortools), 1107
dup_zz_hensel_lift() (in module sympy.polys.factortools), 1106
dup_zz_hensel_step() (in module sympy.polys.factortools), 1105
dup_zz_irreducible_p() (in module sympy.polys.factortools), 1106
dup_zz_zassenhaus() (in module sympy.polys.factortools), 1106
Dyadic (class in sympy.physics.mechanics.essential), 1482
dynamicsymbols() (in module sympy.physics.mechanics.essential), 1484

E

E (in module sympy.core.numbers), 126
e (mpmath.mp attribute), 657
E() (in module sympy.stats), 1252
e1() (in module mpmath), 704
E1() (in module sympy.functions.special.error_functions), 344
E_n() (in module sympy.physics.qho_1d), 1396
E_nl() (in module sympy.physics.hydrogen), 1392
E_nl() (in module sympy.physics.sho), 1397
E_nl_dirac() (in module sympy.physics.hydrogen), 1393
EC() (sympy.polys.polytools.Poly method), 965
eccentricity (sympy.geometry.ellipse.Ellipse attribute), 467
edges (sympy.combinatorics.polyhedron.Polyhedron attribute), 229
edges() (sympy.combinatorics.prufer.Prufer static method), 231
Ei (class in sympy.functions.special.error_functions), 341
ei() (in module mpmath), 702
eigenvals() (sympy.matrices.matrices.MatrixBase method), 567
eigenvects() (sympy.matrices.matrices.MatrixBase method), 567
Eijk() (in module sympy.functions.special.tensor_functions), 394
eject() (sympy.polys.polyclasses.DMP method), 1045
eject() (sympy.polys.polytools.Poly method), 974
ellipe() (in module mpmath), 820
ellipf() (in module mpmath), 818
ellipfun() (in module mpmath), 832
ellipk() (in module mpmath), 816
ellippi() (in module mpmath), 823
elliprc() (in module mpmath), 827
elliprd() (in module mpmath), 829
elliprf() (in module mpmath), 825
elliprg() (in module mpmath), 829
elliprj() (in module mpmath), 828
Ellipse (class in sympy.geometry.ellipse), 464
elliptic_e (class in sympy.functions.special.elliptic_integrals), 379
elliptic_f (class in sympy.functions.special.elliptic_integrals), 379
elliptic_k (class in sympy.functions.special.elliptic_integrals), 378
elliptic_pi (class in sympy.functions.special.elliptic_integrals), 380
EM() (sympy.polys.polytools.Poly method), 966
emptyPrinter() (sympy.printing.repr.ReprPrinter method), 1124
EmptySet (class in sympy.core.sets), 1179
encloses() (sympy.geometry.entity.GeometryEntity method), 436
encloses_point() (sympy.geometry.ellipse.Ellipse method), 467
encloses_point() (sympy.geometry.polygon.Polygon method), 481
encloses_point() (sympy.geometry.polygon.RegularPolygon method), 487
end (sympy.core.sets.Interval attribute), 1175
enumerate_states() (in module sympy.physics.quantum.represent), 1531
EPath (class in sympy.simplify.epathtools), 1202
epath() (in module sympy.simplify.epathtools), 1203
Eq (class in sympy.core.relational), 134
Equality (class in sympy.core.relational), 135

equals() (sympy.core.expr.Expr method), 107
equals() (sympy.matrices.expressions.MatrixExpr method), 615
equals() (sympy.matrices.immutable.ImmutableMatrix method), 613
equation() (sympy.geometry.ellipse.Circle method), 475
equation() (sympy.geometry.ellipse.Ellipse method), 468
equation() (sympy.geometry.line.Line method), 455
Equivalent (class in sympy.logic.boolalg), 545
erf (class in sympy.functions.special.error_functions), 354
erf() (in module mpmath), 710
erf2 (class in sympy.functions.special.error_functions), 358
erf2inv (class in sympy.functions.special.error_functions), 360
erfc (class in sympy.functions.special.error_functions), 356
erfc() (in module mpmath), 711
erfcinv (class in sympy.functions.special.error_functions), 360
erfi (class in sympy.functions.special.error_functions), 357
erfi() (in module mpmath), 711
erfinv (class in sympy.functions.special.error_functions), 359
erfinv() (in module mpmath), 712
Erlang() (in module sympy.stats), 1231
estimate_error() (mpmath.calculus.quadrature.QuadratureRule method), 906
ET() (sympy.polys.polytools.Poly method), 966
eta (sympy.functions.special.hyper.hyper attribute), 375
euler (class in sympy.functions.combinatorial.numbers), 327
euler (mpmath.mp attribute), 657
euler_maclaurin() (sympy.concrete.summations.Sum method), 285
eulernum() (in module mpmath), 863
eulerpoly() (in module mpmath), 864
eval() (sympy.assumptions.assume.Predicate method), 1150
eval() (sympy.functions.special.tensor_functions.Kronecker class method), 395
eval() (sympy.physics.secondquant.Commutator class method), 1410
eval() (sympy.physics.secondquant.Dagger class method), 1398
eval() (sympy.physics.secondquant.KroneckerDelta class method), 1399
eval() (sympy.polys.polyclasses.DMP method), 1045
eval() (sympy.polys.polytools.Poly method), 974
eval_controls() (sympy.physics.quantum.gate.CGate method), 1551
eval_expr() (in module sympy.parsing.sympy_parser), 1379
eval_levicivita() (in module sympy.functions.special.tensor_functions), 394
evalf() (sympy.geometry.point.Point method), 442
evalf() (sympy.matrices.matrices.MatrixBase method), 567
evalf() (sympy.polys.domains.domain.Domain method), 1032
evaluate_deltas() (in module sympy.physics.secondquant), 1414
evaluate_pauli_product() (in module sympy.physics.paulialgebra), 1396
EvaluationFailed (class in sympy.polys.polyerrors), 1112
ExactQuotientFailed (class in sympy.polys.polyerrors), 1111
exclude() (sympy.polys.polyclasses.DMP method), 1045
exclude() (sympy.polys.polytools.Poly method), 974
exp (class in sympy.functions.elementary.exponential), 310
exp() (in module mpmath), 662
exp() (sympy.matrices.matrices.MatrixBase method), 567
expand() (in module sympy.core.function), 157
expand() (sympy.core.expr.Expr method), 107
expand() (sympy.matrices.matrices.MatrixBase method), 567
expand_complex() (in module sympy.core.function), 164
expand_func() (in module sympy.core.function), 164
expand_log() (in module sympy.core.function), 163
expand_mul() (in module sympy.core.function), 163
expand_multinomial() (in module sympy.core.function), 165
expand_power_base() (in module sympy.core.function), 165

expand_power_exp() (in module sympy.core.function), 165
expand_trig() (in module sympy.core.function), 164
expint (class in sympy.functions.special.error_functions), 343
expint() (in module mpmath), 704
expj() (in module mpmath), 663
expjpi() (in module mpmath), 664
expm() (in module mpmath), 922
expm1() (in module mpmath), 664
Exponential() (in module sympy.stats), 1232
Expr (class in sympy.core.expr), 97
expr (sympy.core.function.Lambda attribute), 149
expr (sympy.core.function.Subs attribute), 156
expr (sympy.functions.elementary.piecewise.ExprCondPair attribute), 311
expr (sympy.physics.quantum.operator.DifferentialOperator attribute), 1524
expr (sympy.physics.quantum.state.Wavefunction attribute), 1547
ExprCondPair (class in sympy.functions.elementary.piecewise), 310
express() (in module sympy.physics.mechanics.functions), 1486
express() (sympy.physics.mechanics.essential.Dyadic method), 1483
express() (sympy.physics.mechanics.essential.Vector method), 1481
ExpressionDomain (class in sympy.polys.domains), 1042
ExpressionDomain.Expression (class in sympy.polys.domains), 1042
exquo() (in module sympy.polys.polytools), 953
exquo() (sympy.polys.domains.domain.Domain method), 1032
exquo() (sympy.polys.domains.field.Field method), 1034
exquo() (sympy.polys.domains.ring.Ring method), 1035
exquo() (sympy.polys.polyclasses.DMF method), 1049
exquo() (sympy.polys.polyclasses.DMP method), 1045
exquo() (sympy.polys.polytools.Poly method), 975
exquo_ground() (sympy.polys.polyclasses.DMP method), 1045
exquo_ground() (sympy.polys.polytools.Poly method), 975
extend() (sympy.ntheory.generate.Sieve method), 261
extend() (sympy.plotting.plot.Plot method), 1134
extend_to_no() (sympy.ntheory.generate.Sieve method), 261
exterior_angle (sympy.geometry.polygon.RegularPolygon attribute), 488
extract() (sympy.matrices.matrices.MatrixBase method), 567
extract_additively() (sympy.core.expr.Expr method), 107
extract_branch_factor() (sympy.core.expr.Expr method), 108
extract_leading_order() (sympy.core.add.Add method), 132
extract_multiplicatively() (sympy.core.expr.Expr method), 108
extradps() (in module mpmath), 651
ExtraneousFactors (class in sympy.polys.polyerrors), 1112
extraprec() (in module mpmath), 650
eye() (in module sympy.matrices.dense), 589
eye() (sympy.matrices.sparse.SparseMatrix class method), 603

F

F (in module sympy.physics.secondquant), 1407
F2PyCodeWrapper (class in sympy.utilities.autowrap), 1337
fabs() (in module mpmath), 640
fac2() (in module mpmath), 684
faces (sympy.combinatorics.polyhedron.Polyhedron attribute), 229
factor() (in module sympy.polys.polytools), 961
factor() (sympy.core.expr.Expr method), 109
factor_list() (in module sympy.polys.polytools), 961
factor_list() (sympy.polys.polyclasses.DMP method), 1045
factor_list() (sympy.polys.polytools.Poly method), 975
factor_list_include() (sympy.polys.polyclasses.DMP method), 1045
factor_list_include() (sympy.polys.polytools.Poly method), 975

factor_terms() (in module sympy.core.exprtools), 171
factorial (class in sympy.functions.combinatorial.factorials), 328
factorial() (in module mpmath), 683
factorial() (sympy.polys.domains.FractionField method), 1041
factorial2 (class in sympy.functions.combinatorial.factorials), 329
factorial_notation() (in module sympy.parsing.sympy_parser), 1382
factorint() (in module sympy.ntheory.factor_), 272
factors() (sympy.core.numbers.Rational method), 124
fadd() (in module mpmath), 635
FallingFactorial (class in sympy.functions.combinatorial.factorials), 329
FBra (in module sympy.physics.secondquant), 1407
fcode() (in module sympy.printing.fcode), 1118
FCodeGen (class in sympy.utilities.codegen), 1343
FCodePrinter (class in sympy.printing.fcode), 1119
Fd (in module sympy.physics.secondquant), 1407
fdiff() (sympy.core.function.Function method), 155
fdiff() (sympy.functions.elementary.complexes.Abs method), 304
fdiff() (sympy.functions.elementary.exponential.exp method), 310
fdiff() (sympy.functions.elementary.exponential.LambertW method), 312
fdiff() (sympy.functions.elementary.exponential.log method), 313
fdiff() (sympy.functions.elementary.hyperbolic.sinh method), 319
FDistribution() (in module sympy.stats), 1233
fdiv() (in module mpmath), 638
fdot() (in module mpmath), 639
ff() (in module mpmath), 691
fglm() (sympy.polys.polytools.GroebnerBasis method), 999
fibonacci (class in sympy.functions.combinatorial.numbers), 330
fibonacci() (in module mpmath), 858
Field (class in sympy.polys.domains.field), 1034
field_isomorphism() (in module sympy.polys.numberfields), 1004
fill() (sympy.matrices.dense.MutableDenseMatrix method), 598
fill() (sympy.matrices.sparse.MutableSparseMatrix method), 609
find() (sympy.core.basic.Basic method), 89
find_executable() (in module sympy.utilities.misc), 1367
find_unit() (in module sympy.physics.units), 1426
findpoly() (in module mpmath), 931
findroot() (in module mpmath), 876
FiniteDomain (class in sympy.stats.frv), 1255
FiniteField (class in sympy.polys.domains), 1037
FinitePSpace (class in sympy.stats.frv), 1255
FiniteRV() (in module sympy.stats), 1224
FiniteSet (class in sympy.core.sets), 1176
FisherZ() (in module sympy.stats), 1234
fit() (sympy.statistics.distributions.Normal static method), 1219
fit() (sympy.statistics.distributions.Uniform static method), 1220
FixedBosonicBasis (class in sympy.physics.secondquant), 1408
FKet (in module sympy.physics.secondquant), 1407
FlagError (class in sympy.polys.polyerrors), 1112
FlatMirror (class in sympy.physics.gaussopt), 1386
FlatRefraction (class in sympy.physics.gaussopt), 1386
flatten() (in module sympy.utilities.iterables), 1349
flatten() (sympy.categories.CompositeMorphism method), 1566
flatten() (sympy.core.add.Add class method), 132
flatten() (sympy.core.mul.Mul class method), 129
Float (class in sympy.core.numbers), 121
floor (class in sympy.functions.elementary.integers), 311
floor() (in module mpmath), 642
fmod() (in module mpmath), 639
fmul() (in module mpmath), 637
fneg() (in module mpmath), 636
foci (sympy.geometry.ellipse.Ellipse attribute), 468

FockSpace (class in sympy.physics.quantum.hilbert), 1520
FockState (class in sympy.physics.secondquant), 1406
FockStateBosonBra (class in sympy.physics.secondquant), 1407
FockStateBosonKet (class in sympy.physics.secondquant), 1407
FockStateBra (class in sympy.physics.secondquant), 1406
FockStateKet (class in sympy.physics.secondquant), 1406
focus_distance (sympy.geometry.ellipse.Ellipse attribute), 469
forcing (sympy.physics.mechanics.lagrange.LagrangesMethod attribute), 1507
forcing_full (sympy.physics.mechanics.lagrange.LagrangesMethod attribute), 1507
form_lagranges_equations() (sympy.physics.mechanics.lagrange.LagrangesMethod method), 1507
fourier() (in module mpmath), 913
fourier_transform() (in module sympy.integrals.transforms), 501
fourierval() (in module mpmath), 914
fprod() (in module mpmath), 639
frac (in module sympy.printing.pretty.pretty_symbology), 1129
frac() (in module mpmath), 644
frac_field() (sympy.polys.domains.domain.Domain method), 1032
frac_field() (sympy.polys.domains.FractionField method), 1041
frac_unify() (sympy.polys.polyclasses.DMF method), 1049
fraction() (in module mpmath), 648
fraction() (in module sympy.simplify.simplify), 1190
FractionField (class in sympy.polys.domains), 1040
Frechet() (in module sympy.stats), 1234
free_module() (sympy.polys.domains.ring.Ring method), 1036
free_symbols (sympy.concrete.products.Product attribute), 289
free_symbols (sympy.core.basic.Basic attribute), 89
free_symbols (sympy.functions.elementary.piecewise.ExprCondPair attribute), 311
free_symbols (sympy.geometry.curve.Curve attribute), 462
free_symbols (sympy.integrals.Integral attribute), 531
free_symbols (sympy.physics.quantum.operator.DifferentialOperator attribute), 1525
free_symbols (sympy.polys.polytools.Poly attribute), 976
free_symbols (sympy.polys.polytools.PurePoly attribute), 999
free_symbols_in_domain (sympy.polys.polytools.Poly attribute), 976
FreeModule (class in sympy.polys.agca.modules), 1015
FreeSpace (class in sympy.physics.gaussopt), 1386
fresnelc (class in sympy.functions.special.error_functions), 353
fresnelc() (in module mpmath), 715
FresnelIntegral (class in sympy.functions.special.error_functions), 352
fresnels (class in sympy.functions.special.error_functions), 352
fresnels() (in module mpmath), 714
frexp() (in module mpmath), 647
from_AlgebraicField() (sympy.polys.domains.domain.Domain method), 1032
from_AlgebraicField() (sympy.polys.domains.IntegerRing method), 1038
from_AlgebraicField() (sympy.polys.domains.RationalField method), 1039
from_dict() (sympy.polys.polyclasses.DMP class method), 1045
from_dict() (sympy.polys.polytools.Poly class method), 976
from_expr() (sympy.polys.polytools.Poly class method), 976
from_ExpressionDomain() (sympy.polys.domains.domain.Domain method), 1032
from_ExpressionDomain() (sympy.polys.domains.ExpressionDomain method), 1042
from_FF_gmpy() (sympy.polys.domains.domain.Domain method), 1032
from_FF_gmpy() (sympy.polys.domains.FiniteField method), 1037
from_FF_python() (sympy.polys.domains.domain.Domain method), 1032
from_FF_python() (sympy.polys.domains.FiniteField method), 1037
from_FractionField() (sympy.polys.domains.domain.Domain

method), 1032
from_FractionField() (sympy.polys.domains.ExpressionDomain method), 1042
from_FractionField() (sympy.polys.domains.FractionField method), 1041
from_GlobalPolynomialRing() (sympy.polys.domains.domain.Domain method), 1032
from_GlobalPolynomialRing() (sympy.polys.domains.ExpressionDomain method), 1042
from_GlobalPolynomialRing() (sympy.polys.domains.FractionField method), 1041
from_indices() (sympy.tensor.tensor.TensMul static method), 1333
from_inversion_vector() (sympy.combinatorics.permutations.Permutation class method), 185
from_list() (sympy.polys.polyclasses.DMP class method), 1045
from_list() (sympy.polys.polytools.Poly class method), 976
from_poly() (sympy.polys.polytools.Poly class method), 976
from_PolynomialRing() (sympy.polys.domains.domain.Domain method), 1032
from_QQ_gmpy() (sympy.polys.domains.AlgebraicField method), 1040
from_QQ_gmpy() (sympy.polys.domains.domain.Domain method), 1032
from_QQ_gmpy() (sympy.polys.domains.ExpressionDomain method), 1042
from_QQ_gmpy() (sympy.polys.domains.FiniteField method), 1037
from_QQ_gmpy() (sympy.polys.domains.FractionField method), 1041
from_QQ_python() (sympy.polys.domains.AlgebraicField method), 1040
from_QQ_python() (sympy.polys.domains.domain.Domain method), 1033
from_QQ_python() (sympy.polys.domains.ExpressionDomain method), 1042
from_QQ_python() (sympy.polys.domains.FiniteField method), 1037
from_QQ_python() (sympy.polys.domains.FractionField method), 1041
from_rgs() (sympy.combinatorics.partitions.Partition class method), 173
from_RR_mpmath() (sympy.polys.domains.AlgebraicField method), 1040
from_RR_mpmath() (sympy.polys.domains.domain.Domain method), 1033
from_RR_mpmath() (sympy.polys.domains.ExpressionDomain method), 1042
from_RR_mpmath() (sympy.polys.domains.FiniteField method), 1037
from_RR_mpmath() (sympy.polys.domains.FractionField method), 1041
from_sequence() (sympy.combinatorics.permutations.Permu class method), 185
from_sympy() (sympy.polys.domains.AlgebraicField method), 1040
from_sympy() (sympy.polys.domains.domain.Domain method), 1033
from_sympy() (sympy.polys.domains.ExpressionDomain method), 1042
from_sympy() (sympy.polys.domains.FiniteField method), 1038
from_sympy() (sympy.polys.domains.FractionField method), 1041
from_sympy_list() (sympy.polys.polyclasses.DMP class method), 1045
from_ZZ_gmpy() (sympy.polys.domains.AlgebraicField method), 1040
from_ZZ_gmpy() (sympy.polys.domains.domain.Domain method), 1033
from_ZZ_gmpy() (sympy.polys.domains.ExpressionDomain method), 1042
from_ZZ_gmpy() (sympy.polys.domains.FiniteField method), 1037
from_ZZ_gmpy() (sympy.polys.domains.FractionField method), 1041
from_ZZ_python() (sympy.polys.domains.AlgebraicField method), 1040
from_ZZ_python() (sympy.polys.domains.domain.Domain method), 1033
from_ZZ_python() (sympy.polys.domains.ExpressionDomain method), 1042
from_ZZ_python() (sympy.polys.domains.FiniteField method), 1038
from_ZZ_python() (sympy.polys.domains.FractionField method), 1041
from_Dummy() (sympy.core.basic.Basic class method), 89
fsum() (in module mpmath), 635
fsum() (in module mpmath), 639
full_cyclic_form (sympy.combinatorics.permutations.Permut attribute), 185
fun_eval() (sympy.tensor.tensor.TensAdd method), 1331
fun_eval() (sympy.tensor.tensor.TensMul method), 1333
func (sympy.core.basic.Basic attribute), 89
Function (class in sympy.core.function), 154

function (sympy.integrals.Integral attribute), 531
function (sympy.physics.quantum.operator.DifferentialOperator attribute), 1525
function_exponentiation() (in module sympy.parsing.sympy_parser), 1381
FunctionClass (class in sympy.core.function), 154
FunctionMatrix (class in sympy.matrices.expressions), 617
functions (sympy.geometry.curve.Curve attribute), 462

G

G() (in module sympy.printing.pretty.pretty_symbology), 1128
g() (in module sympy.printing.pretty.pretty_symbology), 1128
gamma (class in sympy.functions.special.gamma_functions), 338
gamma() (in module mpmath), 686
Gamma() (in module sympy.stats), 1235
gammainc() (in module mpmath), 701
GammaInverse() (in module sympy.stats), 1236
gammaprod() (in module mpmath), 688
Gate (class in sympy.physics.quantum.gate), 1550
gate (sympy.physics.quantum.gate.CGate attribute), 1551
gate (sympy.physics.quantum.gate.CNotGate attribute), 1553
gate_simp() (in module sympy.physics.quantum.gate), 1554
gate_sort() (in module sympy.physics.quantum.gate), 1554
gaunt() (in module sympy.physics.wigner), 1419
gauss_chebyshev_t() (in module sympy.integrals.quadrature), 538
gauss_chebyshev_u() (in module sympy.integrals.quadrature), 539
gauss_gen_laguerre() (in module sympy.integrals.quadrature), 537
gauss_hermite() (in module sympy.integrals.quadrature), 537
gauss_jacobi() (in module sympy.integrals.quadrature), 540
gauss_laguerre() (in module sympy.integrals.quadrature), 536
gauss_legendre() (in module sympy.integrals.quadrature), 535
gaussian_conj() (in module sympy.physics.gaussopt), 1390
GaussLegendre (class in mpmath.calculus.quadrature), 908
gcd() (in module sympy.polys.polytools), 957
gcd() (sympy.core.numbers.Number method), 121
gcd() (sympy.polys.domains.domain.Domain method), 1033
gcd() (sympy.polys.domains.field.Field method), 1035
gcd() (sympy.polys.polyclasses.DMP method), 1045
gcd() (sympy.polys.polytools.Poly method), 977
gcd_list() (in module sympy.polys.polytools), 957
gcd_terms() (in module sympy.core.exprtools), 170
gcdex() (in module sympy.polys.polytools), 954
gcdex() (sympy.polys.domains.domain.Domain method), 1033
gcdex() (sympy.polys.polyclasses.DMP method), 1045
gcdex() (sympy.polys.polytools.Poly method), 977
Ge (class in sympy.core.relational), 135
gegenbauer (class in sympy.functions.special.polynomials), 383
gegenbauer() (in module mpmath), 782
gegenbauer_poly() (in module sympy.polys.orthopolys), 1006
gen (sympy.polys.polytools.Poly attribute), 977
generate() (sympy.combinatorics.perm_groups.PermutationG method), 210
generate_bell() (in module sympy.utilities.iterables), 1349
generate_derangements() (in module sympy.utilities.iterables), 1350
generate_dimino() (sympy.combinatorics.perm_groups.Perm method), 211
generate_gray() (sympy.combinatorics.graycode.GrayCode method), 242
generate_involutions() (in module sympy.utilities.iterables), 1350
generate_oriented_forest() (in module sympy.utilities.iterables), 1351
generate_schreier_sims() (sympy.combinatorics.perm_groups.PermutationGro method), 211
generate_tokens() (in module sympy.parsing.sympy_tokenize), 1380

generators (sympy.combinatorics.perm_groups.PermutationGroup attribute), 212
GeneratorsError (class in sympy.polys.polyerrors), 1112
GeneratorsNeeded (class in sympy.polys.polyerrors), 1112
geometric_conj_ab() (in module sympy.physics.gaussopt), 1391
geometric_conj_af() (in module sympy.physics.gaussopt), 1391
geometric_conj_bf() (in module sympy.physics.gaussopt), 1391
GeometricRay (class in sympy.physics.gaussopt), 1386
GeometryEntity (class in sympy.geometry.entity), 436
get() (sympy.core.containers.Dict method), 168
get_adjacency_distance() (sympy.combinatorics.permutations.Permutation method), 185
get_adjacency_matrix() (sympy.combinatorics.permutations.Permutation method), 186
get_basis() (in module sympy.physics.quantum.represent), 1530
get_class() (in module sympy.utilities.source), 1377
get_comm() (sympy.tensor.tensor._TensorManager method), 1325
get_contraction_structure() (in module sympy.tensor.index_methods), 1322
get_default_datatype() (in module sympy.utilities.codegen), 1341
get_diag_blocks() (sympy.matrices.matrices.MatrixBase method), 568
get_domain() (sympy.polys.polytools.Poly method), 977
get_exact() (sympy.polys.domains.domain.Domain method), 1033
get_field() (sympy.polys.domains.domain.Domain method), 1033
get_field() (sympy.polys.domains.ExpressionDomain method), 1042
get_field() (sympy.polys.domains.field.Field method), 1035
get_field() (sympy.polys.domains.FiniteField method), 1038
get_field() (sympy.polys.domains.IntegerRing method), 1038
get_indices() (in module sympy.tensor.index_methods), 1324
get_indices() (sympy.tensor.tensor.TensMul method), 1333
get_interface() (sympy.utilities.codegen.FCodeGen method), 1344
get_mass() (sympy.physics.mechanics.particle.Particle method), 1493
get_mod_func() (in module sympy.utilities.source), 1378
get_modulus() (sympy.polys.polytools.Poly method), 977
get_nodes() (mpmath.calculus.quadrature.QuadratureRule method), 907
get_period() (sympy.functions.special.hyper.meijerg method), 378
get_permuted() (sympy.physics.secondquant.PermutationOp method), 1417
get_point() (sympy.physics.mechanics.particle.Particle method), 1493
get_positional_distance() (sympy.combinatorics.permutations.Permutation method), 186
get_precedence_distance() (sympy.combinatorics.permutations.Permutation method), 187
get_precedence_matrix() (sympy.combinatorics.permutations.Permutation method), 187
get_prototype() (sympy.utilities.codegen.CCodeGen method), 1343
get_resource() (in module sympy.utilities.pkgdata), 1369
get_ring() (sympy.polys.domains.AlgebraicField method), 1040
get_ring() (sympy.polys.domains.domain.Domain method), 1033
get_ring() (sympy.polys.domains.ExpressionDomain method), 1043
get_ring() (sympy.polys.domains.field.Field method), 1035
get_ring() (sympy.polys.domains.FractionField method), 1041
get_ring() (sympy.polys.domains.ring.Ring method), 1036
get_segments() (sympy.plotting.plot.LineOver1DRangeSerie method), 1143
get_segments() (sympy.plotting.plot.Parametric2DLineSerie method), 1143
get_subNO() (sympy.physics.secondquant.NO method), 1412
get_subset_from_bitstring() (sympy.combinatorics.graycode static method), 245
get_symmetric_group_sgs() (in module sympy.combinatorics.tensor_can),

260
get_sympy_dir() (in module sympy.utilities.runtests), 1373
get_target_matrix() (sympy.physics.quantum.gate.Gate method), 1550
get_target_matrix() (sympy.physics.quantum.gate.UGate method), 1551
getn() (sympy.core.expr.Expr method), 109
getO() (sympy.core.expr.Expr method), 109
gf_add() (in module sympy.polys.galoistools), 1083
gf_add_ground() (in module sympy.polys.galoistools), 1082
gf_add_mul() (in module sympy.polys.galoistools), 1084
gf_berlekamp() (in module sympy.polys.galoistools), 1092
gf_cofactors() (in module sympy.polys.galoistools), 1087
gf_compose() (in module sympy.polys.galoistools), 1089
gf_compose_mod() (in module sympy.polys.galoistools), 1089
gf_crt() (in module sympy.polys.galoistools), 1079
gf_crt1() (in module sympy.polys.galoistools), 1079
gf_crt2() (in module sympy.polys.galoistools), 1079
gf_csolve() (in module sympy.polys.galoistools), 1094
gf_degree() (in module sympy.polys.galoistools), 1080
gf_diff() (in module sympy.polys.galoistools), 1088
gf_div() (in module sympy.polys.galoistools), 1085
gf_eval() (in module sympy.polys.galoistools), 1089
gf_expand() (in module sympy.polys.galoistools), 1084
gf_exquo() (in module sympy.polys.galoistools), 1086
gf_factor() (in module sympy.polys.galoistools), 1093
gf_factor_sqf() (in module sympy.polys.galoistools), 1093
gf_from_dict() (in module sympy.polys.galoistools), 1081
gf_from_int_poly() (in module sympy.polys.galoistools), 1081
gf_gcd() (in module sympy.polys.galoistools), 1087
gf_gcdex() (in module sympy.polys.galoistools), 1088
gf_int() (in module sympy.polys.galoistools), 1080
gf_irreducible() (in module sympy.polys.galoistools), 1090
gf_irreducible_p() (in module sympy.polys.galoistools), 1090
gf_LC() (in module sympy.polys.galoistools), 1080
gf_lcm() (in module sympy.polys.galoistools), 1087
gf_lshift() (in module sympy.polys.galoistools), 1086
gf_monic() (in module sympy.polys.galoistools), 1088
gf_mul() (in module sympy.polys.galoistools), 1084
gf_mul_ground() (in module sympy.polys.galoistools), 1083
gf_multi_eval() (in module sympy.polys.galoistools), 1089
gf_neg() (in module sympy.polys.galoistools), 1082
gf_normal() (in module sympy.polys.galoistools), 1081
gf_pow() (in module sympy.polys.galoistools), 1086
gf_pow_mod() (in module sympy.polys.galoistools), 1087
gf_Qbasis() (in module sympy.polys.galoistools), 1092
gf_Qmatrix() (in module sympy.polys.galoistools), 1092
gf_quo() (in module sympy.polys.galoistools), 1085
gf_quo_ground() (in module sympy.polys.galoistools), 1083
gf_random() (in module sympy.polys.galoistools), 1090
gf_rem() (in module sympy.polys.galoistools), 1085
gf_rshift() (in module sympy.polys.galoistools), 1086
gf_shoup() (in module sympy.polys.galoistools), 1093
gf_sqf_list() (in module sympy.polys.galoistools), 1091
gf_sqf_p() (in module sympy.polys.galoistools), 1091
gf_sqf_part() (in module sympy.polys.galoistools), 1091
gf_sqr() (in module sympy.polys.galoistools), 1084

gf_strip() (in module sympy.polys.galoistools), 1080
gf_sub() (in module sympy.polys.galoistools), 1083
gf_sub_ground() (in module sympy.polys.galoistools), 1082
gf_sub_mul() (in module sympy.polys.galoistools), 1084
gf_TC() (in module sympy.polys.galoistools), 1080
gf_to_dict() (in module sympy.polys.galoistools), 1081
gf_to_int_poly() (in module sympy.polys.galoistools), 1082
gf_trace_map() (in module sympy.polys.galoistools), 1090
gf_trunc() (in module sympy.polys.galoistools), 1081
gf_value() (in module sympy.polys.galoistools), 1094
gf_zassenhaus() (in module sympy.polys.galoistools), 1093
gff() (in module sympy.polys.polytools), 960
gff_list() (in module sympy.polys.polytools), 959
gff_list() (sympy.polys.polyclasses.DMP method), 1045
gff_list() (sympy.polys.polytools.Poly method), 978
given() (in module sympy.stats), 1253
glaisher (mpmath.mp attribute), 657
GMPYFiniteField (class in sympy.polys.domains), 1043
GMPYIntegerRing (class in sympy.polys.domains), 1043
GMPYRationalField (class in sympy.polys.domains), 1043
gosper_normal() (in module sympy.concrete.gosper), 291
gosper_sum() (in module sympy.concrete.gosper), 292
gosper_term() (in module sympy.concrete.gosper), 292
gouy (sympy.physics.gaussopt.BeamParameter attribute), 1383
Grad() (in module sympy.galgebra.ga), 418
grampoint() (in module mpmath), 846
GramSchmidt() (in module sympy.matrices.dense), 591
gray_to_bin() (sympy.combinatorics.graycode static method), 244
GrayCode (class in sympy.combinatorics.graycode), 241
graycode_subsets() (sympy.combinatorics.graycode static method), 245
GreaterThan (class in sympy.core.relational), 136
greek_letters (in module sympy.printing.pretty.pretty_symbology), 1128
groebner() (in module sympy.polys.groebnertools), 1109
groebner() (in module sympy.polys.polytools), 964
GroebnerBasis (class in sympy.polys.polytools), 999
ground_roots() (in module sympy.polys.polytools), 963
ground_roots() (sympy.polys.polytools.Poly method), 978
group() (in module sympy.parsing.sympy_tokenize), 1380
group() (in module sympy.utilities.iterables), 1351
grover_iteration() (in module sympy.physics.quantum.grover), 1556
gruntz() (in module sympy.series.gruntz), 1166
Gt (class in sympy.core.relational), 135
guess_degree() (mpmath.calculus.quadrature.QuadratureRule method), 907

H

H (in module sympy.physics.quantum.gate), 1553
H (sympy.matrices.matrices.MatrixBase attribute), 556
HadamardGate (class in sympy.physics.quantum.gate), 1552
half_gcdex() (in module sympy.polys.polytools), 954
half_gcdex() (sympy.polys.domains.domain.Domain method), 1033
half_gcdex() (sympy.polys.polyclasses.DMP method), 1045
half_gcdex() (sympy.polys.polytools.Poly method), 978
half_per() (sympy.polys.polyclasses.DMF method), 1049
Halley (class in mpmath.calculus.optimization), 880
hankel1 (class in sympy.functions.special.bessel), 364
hankel1() (in module mpmath), 730

hankel2 (class in sympy.functions.special.bessel), 365
hankel2() (in module mpmath), 732
hankel_transform() (in module sympy.integrals.transforms), 504
harmonic (class in sympy.functions.combinatorial.numbers), 331
harmonic() (in module mpmath), 699
has() (sympy.core.basic.Basic method), 90
has() (sympy.matrices.matrices.MatrixBase method), 568
has() (sympy.matrices.sparse.SparseMatrix method), 603
has_dups() (in module sympy.utilities.iterables), 1351
has_only_gens() (sympy.polys.polytools.Poly method), 978
has_q_annihilators (sympy.physics.secondquant.NO attribute), 1412
has_q_creators (sympy.physics.secondquant.NO attribute), 1413
has_variety() (in module sympy.utilities.iterables), 1352
Heaviside (class in sympy.functions.special.delta_functions), 337
height (sympy.categories.diagram_drawing.DiagramGrid attribute), 1573
height (sympy.physics.gaussopt.GeometricRay attribute), 1387
height() (sympy.polys.agca.ideals.Ideal method), 1021
height() (sympy.printing.pretty.stringpict.stringPict method), 1130
hermite (class in sympy.functions.special.polynomials), 388
hermite() (in module mpmath), 783
hermite_poly() (in module sympy.polys.orthopolys), 1006
HermitianOperator (class in sympy.physics.quantum.operator), 1522
hessian() (in module sympy.matrices.dense), 590
heurisch() (sympy.integrals static method), 527
heurisch_wrapper() (sympy.integrals static method), 528
HeuristicGCDFailed (class in sympy.polys.polyerrors), 1112
HilbertSpace (class in sympy.physics.quantum.hilbert), 1519
hobj() (in module sympy.printing.pretty.pretty_symbology), 1129
hom() (sympy.categories.Diagram method), 1569
homogeneous_order() (in module sympy.solvers.ode), 1261
homogeneous_order() (sympy.polys.polyclasses.DMP method), 1045
homogeneous_order() (sympy.polys.polytools.Poly method), 979
homomorphism() (in module sympy.polys.agca.homomorphisms), 1026
HomomorphismFailed (class in sympy.polys.polyerrors), 1112
horner() (in module sympy.polys.polyfuncs), 1001
hradius (sympy.geometry.ellipse.Ellipse attribute), 469
hstack() (sympy.matrices.matrices.MatrixBase class method), 569
hyp0f1() (in module mpmath), 793
hyp1f1() (in module mpmath), 794
hyp1f2() (in module mpmath), 795
hyp2f0() (in module mpmath), 796
hyp2f1() (in module mpmath), 797
hyp2f2() (in module mpmath), 798
hyp2f3() (in module mpmath), 798
hyp3f2() (in module mpmath), 799
hyper (class in sympy.functions.special.hyper), 374
hyper() (in module mpmath), 800
hyper2d() (in module mpmath), 807
HyperbolicFunction (class in sympy.functions.elementary.hyperbolic), 311
hypercomb() (in module mpmath), 802
hyperexpand() (in module sympy.simplify.hyperexpand), 1201
hyperfac() (in module mpmath), 694
Hypergeometric() (in module sympy.stats), 1224
hypersimilar() (in module sympy.simplify.simplify), 1195
hypersimp() (in module sympy.simplify.simplify), 1194
hyperu() (in module mpmath), 767
hypot() (in module mpmath), 658

I

I (in module sympy.core.numbers), 126

ibin() (in module sympy.utilities.iterables), 1352
Ideal (class in sympy.polys.agca.ideals), 1021
ideal() (sympy.polys.domains.ring.Ring method), 1036
identify() (in module mpmath), 928
Identity (class in sympy.matrices.expressions), 618
identity_hom() (sympy.polys.agca.modules.FreeModule method), 1016
identity_hom() (sympy.polys.agca.modules.Module method), 1015
identity_hom() (sympy.polys.agca.modules.QuotientModule method), 1023
identity_hom() (sympy.polys.agca.modules.SubModule method), 1017
IdentityFunction (class in sympy.functions.elementary.miscellaneous), 311
IdentityGate (class in sympy.physics.quantum.gate), 1552
IdentityMorphism (class in sympy.categories), 1566
Idx (class in sympy.tensor.indexed), 1317
igcd() (in module sympy.core.numbers), 125
ilcm() (in module sympy.core.numbers), 125
Illinois (class in mpmath.calculus.optimization), 881
im() (in module mpmath), 641
image() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 1028
ImmutableMatrix (class in sympy.matrices.immutable), 612
ImmutableSparseMatrix (class in sympy.matrices.immutable), 611
implemented_function() (in module sympy.utilities.lambdify), 1365
implicit_application() (in module sympy.parsing.sympy_parser), 1381
implicit_multiplication() (in module sympy.parsing.sympy_parser), 1381
implicit_multiplication_application() (in module sympy.parsing.sympy_parser), 1381
ImplicitSeries (class in sympy.plotting.plot_implicit), 1144
Implies (class in sympy.logic.boolalg), 545
in_terms_of_generators() (sympy.polys.agca.modules.SubModule method), 1017
incenter (sympy.geometry.polygon.Triangle attribute), 494
incircle (sympy.geometry.polygon.RegularPolygon attribute), 488
incircle (sympy.geometry.polygon.Triangle attribute), 494
inclusion_hom() (sympy.polys.agca.modules.SubModule method), 1017
indent_code() (sympy.printing.ccode.CCodePrinter method), 1117
indent_code() (sympy.printing.fcode.FCodePrinter method), 1119
index() (sympy.combinatorics.permutations.Permutation method), 187
index() (sympy.physics.secondquant.FixedBosonicBasis method), 1409
index() (sympy.physics.secondquant.VarBosonicBasis method), 1408
Indexed (class in sympy.tensor.indexed), 1318
IndexedBase (class in sympy.tensor.indexed), 1320
indices (sympy.tensor.indexed.Indexed attribute), 1319
indices_contain_equal_information (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 395
indices_contain_equal_information (sympy.physics.secondquant.KroneckerDelta attribute), 1400
inertia() (in module sympy.physics.mechanics.functions), 1498
inertia_of_point_mass() (in module sympy.physics.mechanics.functions), 1499
inf (sympy.core.sets.Set attribute), 1173
infinitesimals() (in module sympy.solvers.ode), 1262
inject() (sympy.polys.domains.compositedomain.CompositeDomain method), 1037
inject() (sympy.polys.domains.domain.Domain method), 1033
inject() (sympy.polys.domains.simpledomain.SimpleDomain method), 1037
inject() (sympy.polys.polyclasses.DMP method), 1046
inject() (sympy.polys.polytools.Poly method), 979
InnerProduct (class in sympy.physics.quantum.innerproduct), 1515
InnerProduct (class in sympy.physics.secondquant), 1407
inradius (sympy.geometry.polygon.RegularPolygon attribute), 488
inradius (sympy.geometry.polygon.Triangle attribute), 494
Integer (class in sympy.core.numbers), 125

integer_nthroot() (in module sympy.core.power), 128
IntegerPartition (class in sympy.combinatorics.partitions), 174
IntegerRing (class in sympy.polys.domains), 1038
Integers (class in sympy.sets.fancysets), 1181
Integral (class in sympy.integrals), 529
Integral.is_commutative (in module sympy.integrals.transforms), 529
integral_steps() (sympy.integrals.manualintegrate static method), 529
integrand() (sympy.functions.special.hyper.meijerg method), 378
integrate() (sympy.core.expr.Expr method), 109
integrate() (sympy.integrals static method), 523
integrate() (sympy.matrices.matrices.MatrixBase method), 569
integrate() (sympy.polys.polyclasses.DMP method), 1046
integrate() (sympy.polys.polytools.Poly method), 979
integrate_result() (in module sympy.physics.quantum.represent), 1530
interactive_traversal() (in module sympy.utilities.iterables), 1353
interior_angle (sympy.geometry.polygon.RegularPolygon attribute), 489
interpolate() (in module sympy.polys.polyfuncs), 1002
interpolating_poly() (in module sympy.polys.specialpolys), 1005
intersect() (sympy.core.sets.Set method), 1173
intersect() (sympy.polys.agca.ideals.Ideal method), 1021
intersect() (sympy.polys.agca.modules.SubModule method), 1018
Intersection (class in sympy.core.sets), 1178
intersection() (in module sympy.geometry.util), 438
intersection() (sympy.geometry.ellipse.Circle method), 476
intersection() (sympy.geometry.ellipse.Ellipse method), 469
intersection() (sympy.geometry.entity.GeometryEntity method), 436
intersection() (sympy.geometry.line.LinearEntity method), 448
intersection() (sympy.geometry.point.Point method), 442
intersection() (sympy.geometry.polygon.Polygon method), 481
Interval (class in sympy.core.sets), 1174
intervals() (in module sympy.polys.polytools), 962
intervals() (sympy.polys.polyclasses.DMP method), 1046
intervals() (sympy.polys.polytools.Poly method), 980
IntQubit (class in sympy.physics.quantum.qubit), 1558
IntQubitBra (class in sympy.physics.quantum.qubit), 1559
Inverse (class in sympy.matrices.expressions), 616
inverse() (sympy.functions.elementary.exponential.log method), 313
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 305
inverse() (sympy.functions.elementary.hyperbolic.acoth method), 305
inverse() (sympy.functions.elementary.hyperbolic.asinh method), 306
inverse() (sympy.functions.elementary.hyperbolic.atanh method), 307
inverse() (sympy.functions.elementary.hyperbolic.coth method), 310
inverse() (sympy.functions.elementary.hyperbolic.sinh method), 320
inverse() (sympy.functions.elementary.hyperbolic.tanh method), 322
inverse() (sympy.functions.elementary.trigonometric.acos method), 305
inverse() (sympy.functions.elementary.trigonometric.acot method), 305
inverse() (sympy.functions.elementary.trigonometric.asin method), 306
inverse() (sympy.functions.elementary.trigonometric.atan method), 307
inverse() (sympy.functions.elementary.trigonometric.cot method), 309
inverse() (sympy.functions.elementary.trigonometric.tan method), 322
inverse_ADJ() (sympy.matrices.matrices.MatrixBase method), 569
inverse_cosine_transform() (in module sympy.integrals.transforms), 504
inverse_fourier_transform() (in module sympy.integrals.transforms), 502
inverse_GE() (sympy.matrices.matrices.MatrixBase method), 569

inverse_hankel_transform() (in module sympy.integrals.transforms), 505
inverse_laplace_transform() (in module sympy.integrals.transforms), 500
inverse_LU() (sympy.matrices.matrices.MatrixBase method), 569
inverse_mellin_transform() (in module sympy.integrals.transforms), 499
inverse_sine_transform() (in module sympy.integrals.transforms), 503
inversion_vector() (sympy.combinatorics.permutations.Permutation method), 188
inversions() (sympy.combinatorics.permutations.Permutation method), 188
invert() (in module sympy.polys.polytools), 954
invert() (sympy.core.expr.Expr method), 109
invert() (sympy.polys.domains.domain.Domain method), 1033
invert() (sympy.polys.domains.ring.Ring method), 1036
invert() (sympy.polys.polyclasses.DMF method), 1049
invert() (sympy.polys.polyclasses.DMP method), 1046
invert() (sympy.polys.polytools.Poly method), 980
IQFT (class in sympy.physics.quantum.qft), 1557
is_abelian (sympy.combinatorics.perm_groups.PermutationGroup attribute), 212
is_above_fermi (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 396
is_above_fermi (sympy.physics.secondquant.KroneckerDelta attribute), 1400
is_algebraic_expr() (sympy.core.expr.Expr method), 109
is_aliased (sympy.polys.numberfields.AlgebraicNumber attribute), 1004
is_alt_sym() (sympy.combinatorics.perm_groups.PermutationGroup method), 212
is_anti_symmetric() (sympy.matrices.matrices.MatrixBase method), 569
is_below_fermi (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 396
is_below_fermi (sympy.physics.secondquant.KroneckerDelta attribute), 1401
is_cnf() (in module sympy.logic.boolalg), 545
is_collinear() (sympy.geometry.point.Point method), 442
is_commutative (sympy.core.function.Function attribute), 155
is_commutative (sympy.physics.quantum.state.Wavefunction attribute), 1547
is_comparable (sympy.core.basic.Basic attribute), 90
is_concurrent() (sympy.geometry.line.LinearEntity method), 449
is_concyclic() (sympy.geometry.point.Point method), 443
is_constant() (sympy.core.expr.Expr method), 110
is_convex() (sympy.geometry.polygon.Polygon method), 482
is_cyclotomic (sympy.polys.polyclasses.DMP attribute), 1046
is_cyclotomic (sympy.polys.polytools.Poly attribute), 980
is_diagonal() (sympy.matrices.matrices.MatrixBase method), 570
is_diagonalizable() (sympy.matrices.matrices.MatrixBase method), 571
is_dnf() (in module sympy.logic.boolalg), 546
is_Empty (sympy.combinatorics.permutations.Permutation attribute), 189
is_equilateral() (sympy.geometry.polygon.Triangle method), 495
is_even (sympy.combinatorics.permutations.Permutation attribute), 190
is_full_module() (sympy.polys.agca.modules.SubModule method), 1018
is_full_module() (sympy.polys.agca.modules.SubQuotientMo method), 1025
is_groebner() (in module sympy.polys.groebnertools), 1110
is_ground (sympy.polys.polyclasses.ANP attribute), 1050
is_ground (sympy.polys.polyclasses.DMP attribute), 1046
is_ground (sympy.polys.polytools.Poly attribute), 980
is_group() (sympy.combinatorics.perm_groups.PermutationG method), 213
is_homogeneous (sympy.polys.polyclasses.DMP attribute), 1046
is_homogeneous (sympy.polys.polytools.Poly attribute), 981
is_identity (sympy.combinatorics.permutations.Permutation attribute), 189
is_identity (sympy.core.function.Lambda attribute), 149
is_injective() (sympy.polys.agca.homomorphisms.ModuleHo method), 1028
is_irreducible (sympy.polys.polyclasses.DMP attribute), 1046
is_irreducible (sympy.polys.polytools.Poly attribute), 981

is_isomorphism() (sympy.polys.agca.homomorphisms.HomomorphismsModule method), 1028
is_isosceles() (sympy.geometry.polygon.Triangle method), 495
is_left_unbounded (sympy.core.sets.Interval attribute), 1175
is_linear (sympy.polys.polyclasses.DMP attribute), 1046
is_linear (sympy.polys.polytools.Poly attribute), 981
is_lower (sympy.matrices.matrices.MatrixBase attribute), 572
is_lower_hessenberg (sympy.matrices.matrices.MatrixBase attribute), 573
is_maximal() (sympy.polys.agca.ideals.Ideal method), 1022
is_minimal() (in module sympy.polys.groebnertools), 1110
is_monic (sympy.polys.polyclasses.DMP attribute), 1046
is_monic (sympy.polys.polytools.Poly attribute), 982
is_monomial (sympy.polys.polyclasses.DMP attribute), 1046
is_monomial (sympy.polys.polytools.Poly attribute), 982
is_multivariate (sympy.polys.polytools.Poly attribute), 982
is_negative() (sympy.polys.domains.AlgebraicField method), 1040
is_negative() (sympy.polys.domains.domain.Domain method), 1033
is_negative() (sympy.polys.domains.ExpressionDomain method), 1043
is_negative() (sympy.polys.domains.FractionField method), 1041
is_nilpotent (sympy.combinatorics.perm_groups.PermutationGroup attribute), 214
is_nilpotent() (sympy.matrices.matrices.MatrixBase method), 573
is_nonnegative() (sympy.polys.domains.AlgebraicField method), 1040
is_nonnegative() (sympy.polys.domains.domain.Domain method), 1033
is_nonnegative() (sympy.polys.domains.ExpressionDomain method), 1043
is_nonnegative() (sympy.polys.domains.FractionField method), 1042
is_nonpositive() (sympy.polys.domains.AlgebraicField method), 1040
is_nonpositive() (sympy.polys.domains.domain.Domain method), 1033
is_nonpositive() (sympy.polys.domains.ExpressionDomain method), 1043
is_nonpositive() (sympy.polys.domains.FractionField method), 1042
is_normal() (sympy.combinatorics.perm_groups.PermutationGroup method), 215
is_normalized (sympy.physics.quantum.state.Wavefunction attribute), 1547
is_number (sympy.concrete.products.Product attribute), 289
is_number (sympy.concrete.summations.Sum attribute), 286
is_number (sympy.core.basic.Basic attribute), 90
is_number (sympy.core.expr.Expr attribute), 111
is_number (sympy.integrals.Integral attribute), 531
is_odd (sympy.combinatorics.permutations.Permutation attribute), 190
is_one (sympy.polys.polyclasses.ANP attribute), 1050
is_one (sympy.polys.polyclasses.DMF attribute), 1049
is_one (sympy.polys.polyclasses.DMP attribute), 1046
is_one (sympy.polys.polytools.Poly attribute), 982
is_one() (sympy.polys.domains.domain.Domain method), 1033
is_only_above_fermi (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 396
is_only_above_fermi (sympy.physics.secondquant.KroneckerDelta attribute), 1401
is_only_below_fermi (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 397
is_only_below_fermi (sympy.physics.secondquant.KroneckerDelta attribute), 1401
is_only_q_annihilator (sympy.physics.secondquant.AnnihilateFermion attribute), 1404
is_only_q_annihilator (sympy.physics.secondquant.CreateFermion attribute), 1405
is_only_q_creator (sympy.physics.secondquant.AnnihilateFermion attribute), 1404
is_only_q_creator (sympy.physics.secondquant.CreateFermion attribute), 1405
is_parallel() (sympy.geometry.line.LinearEntity method), 449

is_perpendicular() (sympy.geometry.line.LinearEntity method), 450
is_polynomial() (sympy.core.expr.Expr method), 111
is_positive() (sympy.polys.domains.AlgebraicField method), 1040
is_positive() (sympy.polys.domains.domain.Domain method), 1033
is_positive() (sympy.polys.domains.ExpressionDomain method), 1043
is_positive() (sympy.polys.domains.FractionField method), 1042
is_primary() (sympy.polys.agca.ideals.Ideal method), 1022
is_prime() (sympy.polys.agca.ideals.Ideal method), 1022
is_primitive (sympy.polys.polyclasses.DMP attribute), 1046
is_primitive (sympy.polys.polytools.Poly attribute), 983
is_primitive() (sympy.combinatorics.perm_groups.PermutationGroup method), 215
is_primitive_root() (in module sympy.ntheory.residue_ntheory), 280
is_principal() (sympy.polys.agca.ideals.Ideal method), 1022
is_q_annihilator (sympy.physics.secondquant.AnnihilateFermion attribute), 1404
is_q_annihilator (sympy.physics.secondquant.CreateFermion attribute), 1406
is_q_creator (sympy.physics.secondquant.AnnihilateFermion attribute), 1405
is_q_creator (sympy.physics.secondquant.CreateFermion attribute), 1406
is_quad_residue() (in module sympy.ntheory.residue_ntheory), 281
is_quadratic (sympy.polys.polyclasses.DMP attribute), 1046
is_quadratic (sympy.polys.polytools.Poly attribute), 983
is_radical() (sympy.polys.agca.ideals.Ideal method), 1022
is_rational_function() (sympy.core.expr.Expr method), 112
is_Real (sympy.core.basic.Basic attribute), 90
is_reduced() (in module sympy.polys.groebnertools), 1110
is_right() (sympy.geometry.polygon.Triangle method), 495
is_right_unbounded (sympy.core.sets.Interval attribute), 1175
is_scalene() (sympy.geometry.polygon.Triangle method), 496
is_sequence() (in module sympy.core.compatibility), 169
is_similar() (sympy.geometry.entity.GeometryEntity method), 436
is_similar() (sympy.geometry.line.LinearEntity method), 450
is_similar() (sympy.geometry.polygon.Triangle method), 496
is_simple() (sympy.functions.special.delta_functions.DiracDe method), 336
is_Singleton (sympy.combinatorics.permutations.Permutatio attribute), 189
is_solvable (sympy.combinatorics.perm_groups.Permutation attribute), 215
is_sqf (sympy.polys.polyclasses.DMP attribute), 1046
is_sqf (sympy.polys.polytools.Poly attribute), 983
is_square (sympy.matrices.matrices.MatrixBase attribute), 573
is_subdiagram() (sympy.categories.Diagram method), 1570
is_subgroup() (sympy.combinatorics.perm_groups.Permutati method), 216
is_submodule() (sympy.polys.agca.modules.FreeModule method), 1016
is_submodule() (sympy.polys.agca.modules.Module method), 1015
is_submodule() (sympy.polys.agca.modules.QuotientModule method), 1024
is_submodule() (sympy.polys.agca.modules.SubModule method), 1018
is_surjective() (sympy.polys.agca.homomorphisms.ModuleH method), 1028
is_symbolic() (sympy.matrices.matrices.MatrixBase method), 574
is_symmetric() (sympy.matrices.matrices.MatrixBase method), 574
is_symmetric() (sympy.matrices.sparse.SparseMatrix method), 604
is_tangent() (sympy.geometry.ellipse.Ellipse method), 470
is_transitive() (sympy.combinatorics.perm_groups.Permutati method), 216
is_trivial (sympy.combinatorics.perm_groups.PermutationGr attribute), 217
is_univariate (sympy.polys.polytools.Poly attribute), 983
is_upper (sympy.matrices.matrices.MatrixBase attribute), 575
is_upper_hessenberg (sympy.matrices.matrices.MatrixBase

attribute), 575
is_whole_ring() (sympy.polys.agca.ideals.Ideal method), 1022
is_zero (sympy.concrete.products.Product attribute), 290
is_zero (sympy.concrete.summations.Sum attribute), 286
is_zero (sympy.integrals.Integral attribute), 532
is_zero (sympy.matrices.matrices.MatrixBase attribute), 576
is_zero (sympy.polys.polyclasses.ANP attribute), 1050
is_zero (sympy.polys.polyclasses.DMF attribute), 1049
is_zero (sympy.polys.polyclasses.DMP attribute), 1046
is_zero (sympy.polys.polytools.Poly attribute), 984
is_zero() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 1029
is_zero() (sympy.polys.agca.ideals.Ideal method), 1022
is_zero() (sympy.polys.agca.modules.FreeModule method), 1016
is_zero() (sympy.polys.agca.modules.Module method), 1015
is_zero() (sympy.polys.agca.modules.QuotientModule method), 1024
is_zero() (sympy.polys.agca.modules.SubModule method), 1018
is_zero() (sympy.polys.domains.domain.Domain method), 1033
is_zero_dimensional (sympy.polys.polytools.GroebnerBasis attribute), 1000
is_zero_dimensional() (in module sympy.polys.polytools), 965
isgeneratorfunction() (in module sympy.utilities.runtests), 1373
isinf() (in module mpmath), 645
isint() (in module mpmath), 646
isnan() (in module mpmath), 645
isnormal() (in module mpmath), 646
isolate() (in module sympy.polys.numberfields), 1004
IsomorphismFailed (class in sympy.polys.polyerrors), 1112
isprime() (in module sympy.ntheory.primetest), 279
ITE (class in sympy.logic.boolalg), 545
items() (sympy.core.containers.Dict method), 168
iter_basic_args() (sympy.core.basic.Basic method), 91
iter_q_annihilators() (sympy.physics.secondquant.NO method), 1413
iter_q_creators() (sympy.physics.secondquant.NO method), 1413
iterable() (in module sympy.core.compatibility), 168
iterate_binary() (sympy.combinatorics.subsets.Subset method), 236
iterate_graycode() (sympy.combinatorics.subsets.Subset method), 236

J

j0() (in module mpmath), 719
j1() (in module mpmath), 719
jacobi (class in sympy.functions.special.polynomials), 381
jacobi() (in module mpmath), 781
jacobi_normalized() (in module sympy.functions.special.polynomials), 382
jacobi_poly() (in module sympy.polys.orthopolys), 1006
jacobi_symbol() (in module sympy.ntheory.residue_ntheory), 281
jacobian() (sympy.diffgeom.CoordSystem method), 1583
jacobian() (sympy.matrices.matrices.MatrixBase method), 576
jn (class in sympy.functions.special.bessel), 365
jn_zeros() (in module sympy.functions.special.bessel), 366
jordan_cell() (in module sympy.matrices.dense), 590
jordan_cells() (sympy.matrices.matrices.MatrixBase method), 577
jordan_form() (sympy.matrices.matrices.MatrixBase method), 577
josephus() (sympy.combinatorics.permutations.Permutation class method), 190
jtheta() (in module mpmath), 830
JxBra (class in sympy.physics.quantum.spin), 1535
JxBraCoupled (class in sympy.physics.quantum.spin), 1538
JxKet (class in sympy.physics.quantum.spin), 1535
JxKetCoupled (class in sympy.physics.quantum.spin), 1537

JyBra (class in sympy.physics.quantum.spin), 1535
JyBraCoupled (class in sympy.physics.quantum.spin), 1538
JyKet (class in sympy.physics.quantum.spin), 1535
JyKetCoupled (class in sympy.physics.quantum.spin), 1538
JzBra (class in sympy.physics.quantum.spin), 1537
JzBraCoupled (class in sympy.physics.quantum.spin), 1540
JzKet (class in sympy.physics.quantum.spin), 1535
JzKetCoupled (class in sympy.physics.quantum.spin), 1538

K

kanes_equations() (sympy.physics.mechanics.kane.KanesMethod method), 1504
KanesMethod (class in sympy.physics.mechanics.kane), 1503
kbins() (in module sympy.utilities.iterables), 1353
kei() (in module mpmath), 737
ker() (in module mpmath), 736
kernel() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 1029
Ket (class in sympy.physics.quantum.state), 1542
ket (sympy.physics.quantum.operator.OuterProduct attribute), 1524
ket (sympy.physics.secondquant.InnerProduct attribute), 1408
KetBase (class in sympy.physics.quantum.state), 1542
key2bounds() (sympy.matrices.matrices.MatrixBase method), 578
key2ij() (sympy.matrices.matrices.MatrixBase method), 578
keys() (sympy.core.containers.Dict method), 168
kfrom() (in module mpmath), 815
khinchin (mpmath.mp attribute), 657
killable_index (sympy.functions.special.tensor_functions.KroneckerDelta attribute), 397
killable_index (sympy.physics.secondquant.KroneckerDelta attribute), 1402
kindiffdict() (sympy.physics.mechanics.kane.KanesMethod method), 1505
kinematic_equations() (in module sympy.physics.mechanics.functions), 1492
kinetic_energy() (in module sympy.physics.mechanics.functions), 1501
kinetic_energy() (sympy.physics.mechanics.particle.Particle method), 1493
kinetic_energy() (sympy.physics.mechanics.rigidbody.RigidB method), 1496
kleinj() (in module mpmath), 833
known_functions (in module sympy.printing.ccode), 1117
KroneckerDelta (class in sympy.functions.special.tensor_functions), 394
KroneckerDelta (class in sympy.physics.secondquant), 1399
ksubsets() (sympy.combinatorics.subsets static method), 241
Kumaraswamy() (in module sympy.stats), 1337

L

l1_norm() (sympy.polys.polyclasses.DMP method), 1046
l1_norm() (sympy.polys.polytools.Poly method), 984
L2 (class in sympy.physics.quantum.hilbert), 1520
label (sympy.physics.quantum.state.TimeDepState attribute), 1544
label (sympy.tensor.indexed.Idx attribute), 1318
label (sympy.tensor.indexed.IndexedBase attribute), 1321
LagrangesMethod (class in sympy.physics.mechanics.lagrange), 1506
Lagrangian() (in module sympy.physics.mechanics.functions), 1502
laguerre (class in sympy.functions.special.polynomials), 389
laguerre() (in module mpmath), 785
laguerre_poly() (in module sympy.polys.orthopolys), 1006
Lambda (class in sympy.core.function), 149
LambdaPrinter (class in sympy.printing.lambdarepr), 1121
lambdarepr() (in module sympy.printing.lambdarepr), 1121
lambdastr() (in module sympy.utilities.lambdify), 1365
lambdify() (in module sympy.utilities.lambdify), 1365

LambertW (class in sympy.functions.elementary.exponential), 312
lambertw() (in module mpmath), 667
Laplace() (in module sympy.stats), 1237
laplace_transform() (in module sympy.integrals.transforms), 500
latex() (in module sympy.printing.latex), 1121
LatexPrinter (class in sympy.printing.latex), 1121
LC() (in module sympy.polys.polytools), 951
LC() (sympy.polys.polyclasses.ANP method), 1049
LC() (sympy.polys.polyclasses.DMP method), 1044
LC() (sympy.polys.polytools.Poly method), 966
lcm() (in module sympy.polys.polytools), 957
lcm() (sympy.core.numbers.Number method), 121
lcm() (sympy.polys.domains.domain.Domain method), 1033
lcm() (sympy.polys.domains.field.Field method), 1035
lcm() (sympy.polys.polyclasses.DMP method), 1046
lcm() (sympy.polys.polytools.Poly method), 984
lcm_list() (in module sympy.polys.polytools), 957
ldexp() (in module mpmath), 646
LDLdecomposition() (sympy.matrices.matrices.MatrixBase method), 557
LDLdecomposition() (sympy.matrices.sparse.SparseMatrix method), 601
LDLsolve() (sympy.matrices.matrices.MatrixBase method), 557
Le (class in sympy.core.relational), 135
leadterm() (sympy.core.expr.Expr method), 113
left (sympy.core.sets.Interval attribute), 1175
left() (sympy.printing.pretty.stringpict.stringPict method), 1130
left_open (sympy.core.sets.Interval attribute), 1175
leftslash() (sympy.printing.pretty.stringpict.stringPict method), 1130
legendre (class in sympy.functions.special.polynomials), 386
legendre() (in module mpmath), 774
legendre_poly() (in module sympy.polys.orthopolys), 1006
legendre_symbol() (in module sympy.ntheory.residue_ntheory), 281
legenp() (in module mpmath), 776
legenq() (in module mpmath), 777
length (sympy.geometry.line.LinearEntity attribute), 450
length (sympy.geometry.line.Segment attribute), 459
length (sympy.geometry.point.Point attribute), 444
length (sympy.geometry.polygon.RegularPolygon attribute), 489
length() (sympy.combinatorics.permutations.Permutation method), 191
length() (sympy.polys.polytools.Poly method), 984
lerchphi (class in sympy.functions.special.zeta_functions), 372
lerchphi() (in module mpmath), 848
LessThan (class in sympy.core.relational), 139
LeviCivita (class in sympy.functions.special.tensor_functions), 394
Li (class in sympy.functions.special.error_functions), 346
li (class in sympy.functions.special.error_functions), 345
li() (in module mpmath), 705
lift() (sympy.polys.polyclasses.DMP method), 1046
lift() (sympy.polys.polytools.Poly method), 985
limit (class in sympy.series.limits), 1165
limit() (in module mpmath), 891
limit() (in module sympy.series.limits), 1164
limit() (sympy.core.expr.Expr method), 113
limit() (sympy.matrices.matrices.MatrixBase method), 578
limit_denominator() (sympy.core.numbers.Rational method), 125
limitinf() (in module sympy.series.gruntz), 1167
limits (sympy.geometry.curve.Curve attribute), 463
limits (sympy.integrals.Integral attribute), 532
limits (sympy.physics.quantum.state.Wavefunction attribute), 1547

Line (class in sympy.geometry.line), 454
Line2DBaseSeries (class in sympy.plotting.plot), 1143
Line3DBaseSeries (class in sympy.plotting.plot), 1144
line_integrate() (sympy.integrals static method), 525
linear_momentum() (in module sympy.physics.mechanics.functions), 1499
linear_momentum() (sympy.physics.mechanics.particle.Particle method), 1494
linear_momentum() (sympy.physics.mechanics.rigidbody.RigidBody method), 1497
LinearEntity (class in sympy.geometry.line), 446
linearize() (sympy.physics.mechanics.kane.KanesMethod method), 1505
LineOver1DRangeSeries (class in sympy.plotting.plot), 1143
linspace() (in module mpmath), 649
list() (sympy.combinatorics.permutations.Cycle method), 199
list() (sympy.combinatorics.permutations.Permutation method), 191
list2numpy() (in module sympy.matrices.dense), 593
liupc() (sympy.matrices.sparse.SparseMatrix method), 604
LM() (in module sympy.polys.polytools), 951
LM() (sympy.polys.polytools.Poly method), 966
ln() (in module mpmath), 666
locatenew() (sympy.physics.mechanics.point.Point method), 1488
log (class in sympy.functions.elementary.exponential), 312
log() (in module mpmath), 665
log() (sympy.polys.domains.domain.Domain method), 1033
log() (sympy.polys.domains.IntegerRing method), 1038
log10() (in module mpmath), 667
logcombine() (in module sympy.simplify.simplify), 1199
loggamma (class in sympy.functions.special.gamma_functions), 338
loggamma() (in module mpmath), 689
Logistic() (in module sympy.stats), 1238
logm() (in module mpmath), 925
LogNormal() (in module sympy.stats), 1239
lommels1() (in module mpmath), 741
lommels2() (in module mpmath), 742
lower (sympy.physics.secondquant.AntiSymmetricTensor attribute), 1415
lower (sympy.tensor.indexed.Idx attribute), 1318
lower_central_series() (sympy.combinatorics.perm_groups.PermutationGro method), 217
lower_triangular_solve() (sympy.matrices.matrices.MatrixBase method), 578
lowergamma (class in sympy.functions.special.gamma_functions), 340
lseries() (sympy.core.expr.Expr method), 113
Lt (class in sympy.core.relational), 134
LT() (in module sympy.polys.polytools), 951
LT() (sympy.polys.polytools.Poly method), 966
ltrim() (sympy.polys.polytools.Poly method), 985
lucas (class in sympy.functions.combinatorial.numbers), 332
LUdecomposition() (sympy.matrices.matrices.MatrixBase method), 558
LUdecomposition_Simple() (sympy.matrices.matrices.MatrixBase method), 558
LUdecompositionFF() (sympy.matrices.matrices.MatrixBase method), 558
LUsolve() (sympy.matrices.matrices.MatrixBase method), 558

M

mag() (in module mpmath), 647
magnitude() (sympy.physics.mechanics.essential.Vector method), 1482
major (sympy.geometry.ellipse.Ellipse attribute), 471
make_perm() (sympy.combinatorics.perm_groups.Permutati method), 218
mangoldt() (in module mpmath), 871
Manifold (class in sympy.diffgeom), 1580
Manifold() (in module sympy.galgebra.ga), 417
manualintegrate() (sympy.integrals static method), 528
map() (sympy.polys.domains.domain.Domain method), 1033
mass (sympy.physics.mechanics.particle.Particle attribute), 1494
mass_matrix (sympy.physics.mechanics.lagrange.Lagranges attribute), 1507

mass_matrix_full (sympy.physics.mechanics.lagrange.LagrangeMethod attribute), 1508
MatAdd (class in sympy.matrices.expressions), 616
match() (sympy.core.basic.Basic method), 91
matches() (sympy.core.basic.Basic method), 91
mathematica() (in module sympy.parsing.mathematica), 1380
mathml() (in module sympy.printing.mathml), 1123
mathml_tag() (sympy.printing.mathml.MathMLPrinter method), 1123
MathMLPrinter (class in sympy.printing.mathml), 1123
MatMul (class in sympy.matrices.expressions), 616
MatPow (class in sympy.matrices.expressions), 616
matrix2numpy() (in module sympy.matrices.dense), 593
matrix_fglm() (in module sympy.polys.fglmtools), 1110
matrix_multiply_elementwise() (in module sympy.matrices.dense), 588
matrix_rep() (in module sympy.physics.secondquant), 1410
matrix_to_density() (in module sympy.physics.quantum.qubit), 1559
matrix_to_qubit() (in module sympy.physics.quantum.qubit), 1559
MatrixBase (class in sympy.matrices.matrices), 556
MatrixError (class in sympy.matrices.matrices), 588
MatrixExpr (class in sympy.matrices.expressions), 614
MatrixSymbol (class in sympy.matrices.expressions), 615
Max (class in sympy.functions.elementary.miscellaneous), 314
max() (sympy.combinatorics.permutations.Permutation method), 192
max_div (sympy.combinatorics.perm_groups.PermutationGroup attribute), 218
max_norm() (sympy.polys.polyclasses.DMP method), 1046
max_norm() (sympy.polys.polytools.Poly method), 985
maxcalls() (in module mpmath), 651
Maxwell() (in module sympy.stats), 1240
maybe() (in module sympy.parsing.sympy_tokenize), 1380
MDNewtonRaphson (class in mpmath.calculus.optimization), 882
measure (sympy.core.sets.Set attribute), 1173
measure_all() (in module sympy.physics.quantum.qubit), 1559
measure_all_oneshot() (in module sympy.physics.quantum.qubit), 1561
measure_partial() (in module sympy.physics.quantum.qubit), 1560
measure_partial_oneshot() (in module sympy.physics.quantum.qubit), 1561
mechanics_printing() (in module sympy.physics.mechanics.functions), 1508
medial (sympy.geometry.polygon.Triangle attribute), 496
medians (sympy.geometry.polygon.Triangle attribute), 497
meets() (sympy.series.gruntz.SubsSet method), 1168
meijerg (class in sympy.functions.special.hyper), 376
meijerg() (in module mpmath), 803
mellin_transform() (in module sympy.integrals.transforms), 499
memoize() (in module mpmath), 651
mertens (mpmath.mp attribute), 657
mfrom() (in module mpmath), 815
mgamma() (in module sympy.physics.matrices), 1395
midpoint (sympy.geometry.line.Segment attribute), 460
midpoint() (sympy.geometry.point.Point method), 444
Min (class in sympy.functions.elementary.miscellaneous), 313
min() (sympy.combinatorics.permutations.Permutation method), 192
min_bits (sympy.physics.quantum.gate.CGate attribute), 1551
min_bits (sympy.physics.quantum.gate.CNotGate attribute), 1553
min_bits (sympy.physics.quantum.gate.Gate attribute), 1550
minimal_block() (sympy.combinatorics.perm_groups.PermutationGroup method), 219
minimal_polynomial() (in module sympy.polys.numberfields), 1003
minlex() (in module sympy.utilities.iterables), 1354
minor (sympy.geometry.ellipse.Ellipse attribute), 471

minorEntry() (sympy.matrices.matrices.MatrixBase method), 579
minorMatrix() (sympy.matrices.matrices.MatrixBase method), 579
minpoly() (in module sympy.polys.numberfields), 1003
mlatex() (in module sympy.physics.mechanics.functions), 1509
MNewton (class in mpmath.calculus.optimization), 880
mnorm() (in module mpmath), 919
Mod (class in sympy.core.mod), 133
Module (class in sympy.polys.agca.modules), 1014
module_quotient() (sympy.polys.agca.modules.SubModule method), 1019
ModuleHomomorphism (class in sympy.polys.agca.homomorphisms), 1027
momentum (sympy.physics.quantum.cartesian.PxBra attribute), 1519
momentum (sympy.physics.quantum.cartesian.PxKet attribute), 1519
monic() (in module sympy.polys.polytools), 958
monic() (sympy.polys.polyclasses.DMP method), 1046
monic() (sympy.polys.polytools.Poly method), 985
monitor() (in module mpmath), 651
monoms() (sympy.polys.polyclasses.DMP method), 1046
monoms() (sympy.polys.polytools.Poly method), 986
Morphism (class in sympy.categories), 1563
morphisms (sympy.categories.diagram_drawing.DiagramGrid attribute), 1573
mpmath.functions.elliptic (module), 813
mpmathify() (in module mpmath), 634
mpprint() (in module sympy.physics.mechanics.functions), 1509
mprint() (in module sympy.physics.mechanics.functions), 1508
mr() (in module sympy.ntheory.primetest), 279
mrv() (in module sympy.series.gruntz), 1167
mrv_leadterm() (in module sympy.series.gruntz), 1167
mrv_max1() (in module sympy.series.gruntz), 1167
mrv_max3() (in module sympy.series.gruntz), 1167
Musigma() (in module sympy.physics.matrices), 1395
Mul (class in sympy.core.mul), 128
Mul() (sympy.assumptions.handlers.calculus.AskBoundedHa static method), 1152
Mul() (sympy.assumptions.handlers.calculus.AskInfinitesima static method), 1153
Mul() (sympy.assumptions.handlers.sets.AskAntiHermitianH static method), 1154
Mul() (sympy.assumptions.handlers.sets.AskHermitianHand static method), 1154
Mul() (sympy.assumptions.handlers.sets.AskImaginaryHand static method), 1155
Mul() (sympy.assumptions.handlers.sets.AskIntegerHandler static method), 1155
Mul() (sympy.assumptions.handlers.sets.AskRationalHandle static method), 1155
Mul() (sympy.assumptions.handlers.sets.AskRealHandler static method), 1155
mul() (sympy.polys.domains.domain.Domain method), 1034
mul() (sympy.polys.polyclasses.DMF method), 1049
mul() (sympy.polys.polyclasses.DMP method), 1047
mul() (sympy.polys.polytools.Poly method), 986
mul_ground() (sympy.polys.polyclasses.DMP method), 1047
mul_ground() (sympy.polys.polytools.Poly method), 986
mul_inv() (sympy.combinatorics.permutations.Permutation method), 192
Muller (class in mpmath.calculus.optimization), 880
MultiFactorial (class in sympy.functions.combinatorial.factorials), 333
multinomial_coefficients() (in module sympy.ntheory.multinomial), 278
multinomial_coefficients_iterator() (in module sympy.ntheory.multinomial), 278
multiplicity() (in module sympy.ntheory.factor_), 268
multiply() (sympy.matrices.matrices.MatrixBase method), 579
multiply() (sympy.matrices.sparse.SparseMatrix method), 604
multiply_elementwise() (sympy.matrices.matrices.MatrixBase method), 579

multiply_ideal() (sympy.polys.agca.modules.FreeModule method), 1016
multiply_ideal() (sympy.polys.agca.modules.Module method), 1015
multiply_ideal() (sympy.polys.agca.modules.SubModule method), 1019
multiset() (in module sympy.utilities.iterables), 1355
multiset_combinations() (in module sympy.utilities.iterables), 1355
multiset_partitions() (in module sympy.utilities.iterables), 1355
multiset_permutations() (in module sympy.utilities.iterables), 1357
MultivariatePolynomialError (class in sympy.polys.polyerrors), 1112
MutableDenseMatrix (class in sympy.matrices.dense), 595
MutableSparseMatrix (class in sympy.matrices.sparse), 607

N

N (class in sympy.core.evalf), 167
n (sympy.combinatorics.graycode.GrayCode attribute), 242
N (sympy.physics.quantum.shor.CMod attribute), 1562
n() (sympy.geometry.point.Point method), 444
n() (sympy.matrices.matrices.MatrixBase method), 579
n() (sympy.polys.domains.domain.Domain method), 1034
n_order() (in module sympy.ntheory.residue_ntheory), 280
Nakagami() (in module sympy.stats), 1240
name (sympy.categories.Category attribute), 1568
name (sympy.categories.NamedMorphism attribute), 1564
NamedMorphism (class in sympy.categories), 1564
nan() (in module sympy.core.numbers), 126
Nand (class in sympy.logic.boolalg), 544
nargs (sympy.core.function.Lambda attribute), 149
native_coeffs() (sympy.polys.numberfields.AlgebraicNumber method), 1004
Naturals (class in sympy.sets.fancysets), 1181
ncdf() (in module mpmath), 714
Ne (class in sympy.core.relational), 134
necklaces() (in module sympy.utilities.iterables), 1357
neg() (sympy.polys.domains.domain.Domain method), 1034
neg() (sympy.polys.polyclasses.DMF method), 1049
neg() (sympy.polys.polyclasses.DMP method), 1047
neg() (sympy.polys.polytools.Poly method), 986
new() (sympy.polys.polytools.Poly class method), 987
Newton (class in mpmath.calculus.optimization), 879
next() (sympy.combinatorics.graycode.GrayCode method), 243
next() (sympy.combinatorics.prufer.Prufer method), 232
next() (sympy.printing.pretty.stringpict.stringPict static method), 1130
next_binary() (sympy.combinatorics.subsets.Subset method), 236
next_gray() (sympy.combinatorics.subsets.Subset method), 237
next_lex() (sympy.combinatorics.partitions.IntegerPartition method), 174
next_lex() (sympy.combinatorics.permutations.Permutation method), 192
next_lexicographic() (sympy.combinatorics.subsets.Subset method), 237
next_nonlex() (sympy.combinatorics.permutations.Permutat method), 192
next_trotterjohnson() (sympy.combinatorics.permutations.Permutation method), 193
nextprime() (in module sympy.ntheory.generate), 263
nfloat() (in module sympy.core.function), 166
nint() (in module mpmath), 643
nint_distance() (in module mpmath), 647
nnz() (sympy.matrices.sparse.SparseMatrix method), 605
NO (class in sympy.physics.secondquant), 1411
no_attrs_in_subclass (in module sympy.utilities.decorator), 1345
nodes (sympy.combinatorics.prufer.Prufer attribute), 232
NonSquareMatrixError (class in sympy.matrices.matrices), 588
Nor (class in sympy.logic.boolalg), 544
norm (sympy.physics.quantum.state.Wavefunction attribute), 1548
norm() (in module mpmath), 918

norm() (sympy.matrices.matrices.MatrixBase method), 579
Normal (class in sympy.statistics.distributions), 1218
Normal() (in module sympy.stats), 1241
normal_closure() (sympy.combinatorics.perm_group.PermutationGroup method), 219
normalize() (sympy.physics.mechanics.essential.Vector method), 1482
normalize() (sympy.physics.quantum.state.WavefunctionSpace method), 1548
normalize() (sympy.statistics.distributions.PDF method), 1221
normalized() (in module sympy.physics.quantum.gate), 1554
normalized() (sympy.matrices.matrices.MatrixBase method), 580
NormalPSpace (class in sympy.stats.crv_types), 1256
Not (class in sympy.logic.boolalg), 544
not_point() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1550
NotAlgebraic (class in sympy.polys.polyerrors), 1112
NotInvertible (class in sympy.polys.polyerrors), 1112
NotReversible (class in sympy.polys.polyerrors), 1112
npartitions() (in module sympy.ntheory.partitions_), 279
npdf() (in module mpmath), 713
nprint() (in module mpmath), 634
nprod() (in module mpmath), 889
nqubits (sympy.physics.quantum.gate.CGate attribute), 1551
nqubits (sympy.physics.quantum.gate.Gate attribute), 1550
nroots() (in module sympy.polys.polytools), 963
nroots() (sympy.polys.polytools.Poly method), 987
nseries() (sympy.core.expr.Expr method), 113
nsimplify() (in module sympy.simplify.simplify), 1195
nsimplify() (sympy.core.expr.Expr method), 114
nsolve() (in module sympy.solvers.solvers), 1303
nstr() (in module mpmath), 634
nsum() (in module mpmath), 882
nth() (sympy.polys.polyclasses.DMP method), 1047
nth() (sympy.polys.polytools.Poly method), 987
nth_power_roots_poly() (in module sympy.polys.polytools), 963
nth_power_roots_poly() (sympy.polys.polytools.Poly method), 987
nthroot() (in module sympy.simplify.simplify), 1188
nu (sympy.functions.special.hyper.meijerg attribute), 378
nullspace() (sympy.matrices.matrices.MatrixBase method), 580
Number (class in sympy.core.numbers), 120
numbered_symbols() (in module sympy.utilities.iterables), 1357
NumberSymbol (class in sympy.core.numbers), 125
numer() (sympy.polys.domains.AlgebraicField method), 1040
numer() (sympy.polys.domains.domain.Domain method), 1034
numer() (sympy.polys.domains.ExpressionDomain method), 1043
numer() (sympy.polys.domains.FractionField method), 1042
numer() (sympy.polys.domains.ring.Ring method), 1036
numer() (sympy.polys.polyclasses.DMF method), 1049
nzeros() (in module mpmath), 844

O

Object (class in sympy.categories), 1563
objects (sympy.categories.Category attribute), 1568
objects (sympy.categories.Diagram attribute), 1570
ode_1st_exact() (in module sympy.solvers.ode), 1267
ode_1st_homogeneous_coeff_best() (in module sympy.solvers.ode), 1268
ode_1st_homogeneous_coeff_subs_dep_div_indep() (in module sympy.solvers.ode), 1269
ode_1st_homogeneous_coeff_subs_indep_div_dep() (in module sympy.solvers.ode), 1270
ode_1st_linear() (in module sympy.solvers.ode), 1272
ode_almost_linear() (in module sympy.solvers.ode), 1280
ode_Bernoulli() (in module sympy.solvers.ode), 1272
ode_linear_coefficients() (in module sympy.solvers.ode), 1281
ode_Liouville() (in module sympy.solvers.ode), 1274

ode_nth_linear_constant_coeff_homogeneous() (in module sympy.solvers.ode), 1276
ode_nth_linear_constant_coeff_undetermined_coefficients() (in module sympy.solvers.ode), 1277
ode_nth_linear_constant_coeff_variation_of_parameters() (in module sympy.solvers.ode), 1278
ode_order() (in module sympy.solvers.deutils), 1305
ode_Riccati_special_minus2() (in module sympy.solvers.ode), 1275
ode_separable() (in module sympy.solvers.ode), 1279
ode_separable_reduced() (in module sympy.solvers.ode), 1282
ode_sol_simplicity() (in module sympy.solvers.ode), 1266
odefun() (in module mpmath), 909
odesimp() (in module sympy.solvers.ode), 1264
of_type() (sympy.polys.domains.domain.Domain method), 1034
one (sympy.polys.polytools.Poly attribute), 988
one_qubit_box() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1550
OneQubitGate (class in sympy.physics.quantum.gate), 1551
ones() (in module sympy.matrices.dense), 589
oo() (in module sympy.core.numbers), 126
OperationNotSupported (class in sympy.polys.polyerrors), 1111
Operator (class in sympy.physics.quantum.operator), 1521
operators (sympy.physics.quantum.state.StateBase attribute), 1542
operators_to_state() (in module sympy.physics.quantum.operatorset), 1526
OptionError (class in sympy.polys.polyerrors), 1112
Or (class in sympy.logic.boolalg), 543
OracleGate (class in sympy.physics.quantum.grover), 1555
orbit() (sympy.combinatorics.perm_groups.PermutationGroup method), 220
orbit_rep() (sympy.combinatorics.perm_groups.PermutationGroup method), 220
orbit_transversal() (sympy.combinatorics.perm_groups.PermutationGroup method), 221
orbits() (sympy.combinatorics.perm_groups.PermutationGroup method), 221
Order (class in sympy.series.order), 1169
order (sympy.functions.special.bessel.BesselBase attribute), 362
order() (sympy.combinatorics.perm_groups.PermutationGroup method), 221
order() (sympy.combinatorics.permutations.Permutation method), 193
orient() (sympy.physics.mechanics.essential.ReferenceFrame method), 1477
orientnew() (sympy.physics.mechanics.essential.ReferenceFrame method), 1477
orthocenter (sympy.geometry.polygon.Triangle attribute), 497
outer() (in module sympy.physics.mechanics.functions), 1486
outer() (sympy.physics.mechanics.essential.Vector method), 1482
OuterProduct (class in sympy.physics.quantum.operator), 1522

P

P() (in module sympy.stats), 1251
p1 (sympy.geometry.line.LinearEntity attribute), 451
p2 (sympy.geometry.line.LinearEntity attribute), 451
pade() (in module mpmath), 912
parallel_line() (sympy.geometry.line.LinearEntity method), 451
parallel_poly_from_expr() (in module sympy.polys.polytools), 950
parameter (sympy.geometry.curve.Curve attribute), 463
Parametric2DLineSeries (class in sympy.plotting.plot), 1143
Parametric3DLineSeries (class in sympy.plotting.plot), 1144
ParametricSurfaceSeries (class in sympy.plotting.plot), 1144
parens() (sympy.printing.pretty.stringpict.stringPict method), 1130
Pareto() (in module sympy.stats), 1242
parity() (sympy.combinatorics.permutations.Permutation method), 194
parse_expr() (in module sympy.parsing.sympy_parser), 1378
parse_maxima() (in module sympy.parsing.maxima), 1380
partial_velocity() (in module sympy.physics.mechanics.functions), 1505
Particle (class in sympy.physics.mechanics.particle), 1492

Partition (class in sympy.combinatorics.partitions), 172
partition (sympy.combinatorics.partitions.Partition attribute), 173
partitions() (in module sympy.utilities.iterables), 1358
pat_matrix() (in module sympy.physics.matrices), 1395
Patch (class in sympy.diffgeom), 1580
pcfd() (in module mpmath), 770
pcfu() (in module mpmath), 772
pcfv() (in module mpmath), 772
pcfw() (in module mpmath), 773
pde_1st_linear_constant_coeff() (in module sympy.solvers.pde), 1293
pde_1st_linear_constant_coeff_homogeneous() (in module sympy.solvers.pde), 1292
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
PDF (class in sympy.statistics.distributions), 1221
pdf() (sympy.statistics.distributions.Normal method), 1220
pdf() (sympy.statistics.distributions.Uniform method), 1220
pdiv() (in module sympy.polys.polytools), 952
pdiv() (sympy.polys.polyclasses.DMP method), 1047
pdiv() (sympy.polys.polytools.Poly method), 988
pdsolve() (in module sympy.solvers.pde), 1289
Pegasus (class in mpmath.calculus.optimization), 881
per() (sympy.polys.polyclasses.DMF method), 1049
per() (sympy.polys.polyclasses.DMP method), 1047
per() (sympy.polys.polytools.Poly method), 988
perfect_power() (in module sympy.ntheory.factor_), 268
periapsis (sympy.geometry.ellipse.Ellipse attribute), 472
perimeter (sympy.geometry.polygon.Polygon attribute), 482
period_find() (in module sympy.physics.quantum.shor), 1562
perm2tensor() (sympy.tensor.tensor.TensMul method), 1334
Permutation (class in sympy.combinatorics.permutations), 177
PermutationGroup (class in sympy.combinatorics.perm_groups), 201
PermutationOperator (class in sympy.physics.secondquant), 1417
permute() (sympy.polys.polyclasses.DMP method), 1047
permuteBkwd() (sympy.matrices.matrices.MatrixBase method), 580
permuteFwd() (sympy.matrices.matrices.MatrixBase method), 581
perpendicular_bisector() (sympy.geometry.line.Segment method), 460
perpendicular_line() (sympy.geometry.line.LinearEntity method), 452
perpendicular_segment() (sympy.geometry.line.LinearEntity method), 452
pexquo() (in module sympy.polys.polytools), 952
pexquo() (sympy.polys.polyclasses.DMP method), 1047
pexquo() (sympy.polys.polytools.Poly method), 988
pgroup (sympy.combinatorics.polyhedron.Polyhedron attribute), 229
Phase (in module sympy.physics.quantum.gate), 1554
PhaseGate (class in sympy.physics.quantum.gate), 1552
phi (mpmath.mp attribute), 657
pi (mpmath.mp attribute), 657
pi() (in module sympy.core.numbers), 126
PIABBra (class in sympy.physics.quantum.piab), 1562
PIABHamiltonian (class in sympy.physics.quantum.piab), 1562
PIABKet (class in sympy.physics.quantum.piab), 1562
Piecewise (class in sympy.functions.elementary.piecewise), 315
piecewise_fold() (in module sympy.functions.elementary.piecewise), 316
Plot (class in sympy.plotting.plot), 1132
plot() (in module mpmath), 653

plot() (in module sympy.plotting.plot), 1134
plot3d() (in module sympy.plotting.plot), 1138
plot3d_parametric_line() (in module sympy.plotting.plot), 1139
plot3d_parametric_surface() (in module sympy.plotting.plot), 1140
plot_implicit() (in module sympy.plotting.plot_implicit), 1141
plot_interval() (sympy.geometry.curve.Curve method), 463
plot_interval() (sympy.geometry.ellipse.Ellipse method), 472
plot_interval() (sympy.geometry.line.Line method), 455
plot_interval() (sympy.geometry.line.Ray method), 457
plot_interval() (sympy.geometry.line.Segment method), 460
plot_interval() (sympy.geometry.polygon.Polygon method), 482
plot_parametric() (in module sympy.plotting.plot), 1136
Point (class in sympy.geometry.point), 440
Point (class in sympy.physics.mechanics.point), 1487
point (sympy.core.function.Subs attribute), 156
point (sympy.physics.mechanics.particle.Particle attribute), 1494
point() (sympy.diffgeom.CoordSystem method), 1583
point_to_coords() (sympy.diffgeom.CoordSystem method), 1583
points (sympy.geometry.line.LinearEntity attribute), 452
pointwise_stabilizer() (sympy.combinatorics.perm_groups.PermutationGroup method), 222
polar() (in module mpmath), 642
PoleError (class in sympy.core.function), 162
PolificationFailed (class in sympy.polys.polyerrors), 1112
pollard_pm1() (in module sympy.ntheory.factor_), 270
pollard_rho() (in module sympy.ntheory.factor_), 268
Poly (class in sympy.polys.polytools), 965
poly() (in module sympy.polys.polytools), 950
poly_from_expr() (in module sympy.polys.polytools), 950
poly_ring() (sympy.polys.domains.domain.Domain method), 1034
poly_ring() (sympy.polys.domains.FractionField method), 1042
poly_unify() (sympy.polys.polyclasses.DMF method), 1049
polyexp() (in module mpmath), 854
polygamma (class in sympy.functions.special.gamma_functions), 339
Polygon (class in sympy.geometry.polygon), 477
Polyhedron (class in sympy.combinatorics.polyhedron), 228
polylog (class in sympy.functions.special.zeta_functions), 371
polylog() (in module mpmath), 849
PolynomialError (class in sympy.polys.polyerrors), 1112
PolynomialRing (class in sympy.polys.domains), 1038
polyroots() (in module mpmath), 875
polyval() (in module mpmath), 874
pos() (sympy.polys.domains.domain.Domain method), 1034
pos_from() (sympy.physics.mechanics.point.Point method), 1489
POSform() (in module sympy.logic.boolalg), 542
posify() (in module sympy.simplify.simplify), 1197
position (sympy.physics.quantum.cartesian.XBra attribute), 1518
position (sympy.physics.quantum.cartesian.XKet attribute), 1518
position_x (sympy.physics.quantum.cartesian.PositionState3 attribute), 1519
position_y (sympy.physics.quantum.cartesian.PositionState3 attribute), 1519
position_z (sympy.physics.quantum.cartesian.PositionState3 attribute), 1519
PositionBra3D (class in sympy.physics.quantum.cartesian), 1519
PositionKet3D (class in sympy.physics.quantum.cartesian), 1519
PositionState3D (class in sympy.physics.quantum.cartesian), 1519
postfixes() (in module sympy.utilities.iterables), 1359
postorder_traversal() (in module sympy.utilities.iterables), 1359
potential_energy (sympy.physics.mechanics.particle.Particle attribute), 1494

potential_energy (sympy.physics.mechanics.rigidbody.RigidBody attribute), 1497
potential_energy() (in module sympy.physics.mechanics.functions), 1501
Pow (class in sympy.core.power), 127
Pow() (sympy.assumptions.handlers.calculus.AskPositiveHandler static method), 1153
Pow() (sympy.assumptions.handlers.calculus.AskPrimeHandler static method), 1153
Pow() (sympy.assumptions.handlers.ntheory.AskPreferredHandler static method), 1153
Pow() (sympy.assumptions.handlers.order.AskNegativeHandler static method), 1154
Pow() (sympy.assumptions.handlers.sets.AskAntiHermitianHandler static method), 1154
Pow() (sympy.assumptions.handlers.sets.AskHermitianHandler static method), 1155
Pow() (sympy.assumptions.handlers.sets.AskImaginaryHandler static method), 1155
Pow() (sympy.assumptions.handlers.sets.AskIntegerHandler static method), 1155
Pow() (sympy.assumptions.handlers.sets.AskRationalHandler static method), 1155
Pow() (sympy.assumptions.handlers.sets.AskRealHandler static method), 1155
pow() (sympy.polys.domains.domain.Domain method), 1034
pow() (sympy.polys.polyclasses.ANP method), 1050
pow() (sympy.polys.polyclasses.DMF method), 1049
pow() (sympy.polys.polyclasses.DMP method), 1047
pow() (sympy.polys.polytools.Poly method), 989
powdenest() (in module sympy.simplify.simplify), 1197
power() (in module mpmath), 663
powm() (in module mpmath), 927
powm1() (in module mpmath), 664
powsimp() (in module sympy.simplify.simplify), 1192
powsimp() (sympy.core.expr.Expr method), 114
pprint_nodes() (in module sympy.printing.tree), 1125
pquo() (in module sympy.polys.polytools), 952
pquo() (sympy.polys.polyclasses.DMP method), 1047
pquo() (sympy.polys.polytools.Poly method), 989
PRECEDENCE (in module sympy.printing.precedence), 1128
precedence() (in module sympy.printing.precedence), 1128
PRECEDENCE_FUNCTIONS (in module sympy.printing.precedence), 1128
PRECEDENCE_VALUES (in module sympy.printing.precedence), 1128
PrecisionExhausted (class in sympy.core.evalf), 167
PrefinitesimalHandler (class in sympy.assumptions.assume), 1149
preferred_index (sympy.functions.special.tensor_functions.K attribute), 398
preferred_index (sympy.physics.secondquant.KroneckerDelt attribute), 1402
prefixes() (in module sympy.utilities.iterables), 1360
prem() (in module sympy.polys.polytools), 952
prem() (sympy.polys.polyclasses.DMP method), 1047
prem() (sympy.polys.polytools.Poly method), 989
premises (sympy.categories.Diagram attribute), 1570
pretty() (in module sympy.printing.pretty.pretty), 1116
pretty_atom() (in module sympy.printing.pretty.pretty_symbology), 1129
pretty_print() (in module sympy.printing.pretty.pretty), 1116
pretty_symbol() (in module sympy.printing.pretty.pretty_symbology), 1129
pretty_try_use_unicode() (in module sympy.printing.pretty.pretty_symbology), 1128
pretty_use_unicode() (in module sympy.printing.pretty.pretty_symbology), 1128
prettyForm (class in sympy.printing.pretty.stringpict), 1130
PrettyPrinter (class in sympy.printing.pretty.pretty), 1116
prev() (sympy.combinatorics.prufer.Prufer method), 232
prev_binary() (sympy.combinatorics.subsets.Subset method), 237
prev_gray() (sympy.combinatorics.subsets.Subset method), 238
prev_lex() (sympy.combinatorics.partitions.IntegerPartition method), 175
prev_lexicographic() (sympy.combinatorics.subsets.Subset

method), 238
preview() (in module sympy.printing.preview), 1126
preview_diagram() (in module sympy.categories.diagram_drawing), 1579
prevprime() (in module sympy.ntheory.generate), 263
prime() (in module sympy.ntheory.generate), 262
primefactors() (in module sympy.ntheory.factor_), 274
primepi() (in module mpmath), 867
primepi() (in module sympy.ntheory.generate), 263
primepi2() (in module mpmath), 867
primerange() (in module sympy.ntheory.generate), 264
primerange() (sympy.ntheory.generate.Sieve method), 262
primezeta() (in module mpmath), 855
primitive() (in module sympy.polys.polytools), 958
primitive() (sympy.core.add.Add method), 132
primitive() (sympy.core.expr.Expr method), 114
primitive() (sympy.polys.polyclasses.DMP method), 1047
primitive() (sympy.polys.polytools.Poly method), 989
primitive_element() (in module sympy.polys.numberfields), 1004
primorial() (in module sympy.ntheory.generate), 265
print_ccode() (in module sympy.printing.ccode), 1118
print_fcode() (in module sympy.printing.fcode), 1119
print_gtk() (in module sympy.printing.gtk), 1121
print_latex() (in module sympy.printing.latex), 1123
print_mathml() (in module sympy.printing.mathml), 1124
print_node() (in module sympy.printing.tree), 1125
print_nonzero() (sympy.matrices.matrices.MatrixBase method), 581
print_tree() (in module sympy.printing.tree), 1125
Printer (class in sympy.printing.printer), 1114
printmethod (sympy.printing.ccode.CCodePrinter attribute), 1117
printmethod (sympy.printing.codeprinter.CodePrinter attribute), 1128
printmethod (sympy.printing.fcode.FCodePrinter attribute), 1119
printmethod (sympy.printing.lambdarepr.LambdaPrinter attribute), 1121
printmethod (sympy.printing.latex.LatexPrinter attribute), 1121
printmethod (sympy.printing.mathml.MathMLPrinter attribute), 1123
printmethod (sympy.printing.pretty.pretty.PrettyPrinter attribute), 1116
printmethod (sympy.printing.printer.Printer attribute), 1115
printmethod (sympy.printing.repr.ReprPrinter attribute), 1124
printmethod (sympy.printing.str.StrPrinter attribute), 1124
printtoken() (in module sympy.parsing.sympy_tokenize), 1379
prob() (sympy.physics.quantum.state.Wavefunction method), 1549
probability() (sympy.statistics.distributions.ContinuousProba method), 1218
prod() (in module sympy.core.mul), 130
Product (class in sympy.concrete.products), 286
product() (in module sympy.concrete.products), 291
product() (sympy.polys.agca.ideals.Ideal method), 1022
ProductDomain (class in sympy.stats.rv), 1255
ProductPSpace (class in sympy.stats.rv), 1255
ProductSet (class in sympy.core.sets), 1179
Proj() (in module sympy.galgebra.ga), 418
project() (sympy.matrices.matrices.MatrixBase method), 581
projection() (sympy.geometry.line.LinearEntity method), 453
Prufer (class in sympy.combinatorics.prufer), 231
prufer_rank() (sympy.combinatorics.prufer.Prufer method), 232
prufer_repr (sympy.combinatorics.prufer.Prufer attribute), 233
psi() (in module mpmath), 698
psi_n() (in module sympy.physics.qho_1d), 1396
pslq() (in module mpmath), 933
PSpace (class in sympy.stats.rv), 1255
pspace() (in module sympy.stats.rv), 1256
PurePoly (class in sympy.polys.polytools), 999

PxBra (class in sympy.physics.quantum.cartesian), 1519
PxKet (class in sympy.physics.quantum.cartesian), 1518
PxOp (class in sympy.physics.quantum.cartesian), 1518
PyTestReporter (class in sympy.utilities.runtests), 1371
Python Enhancement Proposals
PEP 335, 1617–1619
PythonFiniteField (class in sympy.polys.domains), 1043
PythonIntegerRing (class in sympy.polys.domains), 1043
PythonRationalField (class in sympy.polys.domains), 1043

Q

Q (class in sympy.assumptions.ask), 1147
q (sympy.physics.gaussopt.BeamParameter attribute), 1384
qapply() (in module sympy.physics.quantum.qapply), 1527
qbarfrom() (in module mpmath), 815
qfac() (in module mpmath), 873
qfrom() (in module mpmath), 814
QFT (class in sympy.physics.quantum.qft), 1557
qgamma() (in module mpmath), 873
qhyper() (in module mpmath), 874
qp() (in module mpmath), 871
QRdecomposition() (sympy.matrices.matrices.MatrixBase method), 559
QRsolve() (sympy.matrices.matrices.MatrixBase method), 559
quad() (in module mpmath), 900
quadosc() (in module mpmath), 904
QuadraticU() (in module sympy.stats), 1243
QuadratureRule (class in mpmath.calculus.quadrature), 906
Qubit (class in sympy.physics.quantum.qubit), 1557
qubit_to_matrix() (in module sympy.physics.quantum.qubit), 1559
QubitBra (class in sympy.physics.quantum.qubit), 1558
quo() (in module sympy.polys.polytools), 953
quo() (sympy.polys.domains.domain.Domain method), 1034
quo() (sympy.polys.domains.field.Field method), 1035
quo() (sympy.polys.domains.ring.Ring method), 1036
quo() (sympy.polys.polyclasses.DMF method), 1049
quo() (sympy.polys.polyclasses.DMP method), 1047
quo() (sympy.polys.polytools.Poly method), 990
quo_ground() (sympy.polys.polyclasses.DMP method), 1047
quo_ground() (sympy.polys.polytools.Poly method), 990
quotient() (sympy.polys.agca.ideals.Ideal method), 1022
quotient_codomain() (sympy.polys.agca.homomorphisms.ModuleHomomo method), 1029
quotient_domain() (sympy.polys.agca.homomorphisms.Modu method), 1029
quotient_hom() (sympy.polys.agca.modules.QuotientModule method), 1024
quotient_hom() (sympy.polys.agca.modules.SubQuotientMod method), 1025
quotient_module() (sympy.polys.agca.modules.FreeModule method), 1016
quotient_module() (sympy.polys.agca.modules.Module method), 1015
quotient_module() (sympy.polys.agca.modules.SubModule method), 1019
quotient_ring() (sympy.polys.domains.ring.Ring method), 1036
QuotientModule (class in sympy.polys.agca.modules), 1023

R

R_nl() (in module sympy.physics.hydrogen), 1393
R_nl() (in module sympy.physics.sho), 1397
racah() (in module sympy.physics.wigner), 1420
rad_rationalize() (in module sympy.simplify.simplify), 1188
radians() (in module mpmath), 673
radical() (sympy.polys.agca.ideals.Ideal method), 1022
radius (sympy.geometry.ellipse.Circle attribute), 476
radius (sympy.geometry.polygon.RegularPolygon attribute), 489
radius (sympy.physics.gaussopt.BeamParameter attribute), 1384
radius_of_convergence (sympy.functions.special.hyper.hyper attribute), 375
radsimp() (in module sympy.simplify.simplify), 1188

radsimp() (sympy.core.expr.Expr method), 114
RaisedCosine() (in module sympy.stats), 1244
raises() (in module sympy.utilities.pytest), 1369
rand() (in module mpmath), 648
randMatrix() (in module sympy.matrices.dense), 592
random() (sympy.combinatorics.perm_groups.PermutationGroup method), 223
random() (sympy.combinatorics.permutations.Permutation class method), 194
random() (sympy.statistics.distributions.ContinuousProbability method), 1218
random_bitstring() (sympy.combinatorics.graycode.GrayCode static method), 244
random_circuit() (in module sympy.physics.quantum.gate), 1554
random_complex_number() (in module sympy.utilities.randtest), 1370
random_integer_partition() (in module sympy.combinatorics.partitions), 175
random_point() (sympy.geometry.ellipse.Ellipse method), 472
random_point() (sympy.geometry.line.LinearEntity method), 453
random_poly() (in module sympy.polys.specialpolys), 1006
random_pr() (sympy.combinatorics.perm_groups.PermutationGroup method), 223
random_stab() (sympy.combinatorics.perm_groups.PermutationGroup method), 223
random_symbols() (in module sympy.stats.rv), 1256
RandomDomain (class in sympy.stats.rv), 1255
RandomSymbol (class in sympy.stats.rv), 1255
randprime() (in module sympy.ntheory.generate), 265
ranges (sympy.tensor.indexed.Indexed attribute), 1319
rank (sympy.combinatorics.graycode.GrayCode attribute), 243
rank (sympy.combinatorics.partitions.Partition attribute), 173
rank (sympy.combinatorics.prufer.Prufer attribute), 233
rank (sympy.tensor.indexed.Indexed attribute), 1320
rank() (sympy.combinatorics.permutations.Permutation method), 194
rank() (sympy.matrices.matrices.MatrixBase method), 582
rank_binary (sympy.combinatorics.subsets.Subset attribute), 238
rank_gray (sympy.combinatorics.subsets.Subset attribute), 238
rank_lexicographic (sympy.combinatorics.subsets.Subset attribute), 239
rank_trotterjohnson() (sympy.combinatorics.permutations.Permutation method), 194
rank_trotterjohnson() (sympy.combinatorics.permutations.Permutation method), 195
rat_clear_denoms() (sympy.polys.polytools.Poly method), 990
ratint() (sympy.integrals static method), 526
Rational (class in sympy.core.numbers), 123
RationalField (class in sympy.polys.domains), 1039
rationalize() (in module sympy.parsing.sympy_parser), 1382
ratsimp() (in module sympy.simplify.simplify), 1190
ratsimp() (sympy.core.expr.Expr method), 114
raylines() (in module sympy.utilities.misc), 1367
Ray (class in sympy.geometry.line), 456
Rayleigh() (in module sympy.stats), 1244
RayleighDistribution() (in module sympy.physics.gaussopt), 1392
RayTransferMatrix (class in sympy.physics.gaussopt), 1388
rcall() (sympy.core.expr.Expr method), 114
rcollect() (in module sympy.simplify.simplify), 1186
re (class in sympy.functions.elementary.complexes), 317
re() (in module mpmath), 641
real_roots() (in module sympy.polys.polytools), 963
real_roots() (sympy.polys.polytools.Poly method), 991
RealNumber (in module sympy.core.numbers), 125
rect() (in module mpmath), 642
recurrence_memo() (in module sympy.utilities.memoization), 1367
red_groebner() (in module sympy.polys.groebnertools), 1109
reduce() (sympy.core.sets.Intersection static method), 1178
reduce() (sympy.core.sets.Union static method), 1177

reduce() (sympy.polys.polytools.GroebnerBasis method), 1000
reduce_element() (sympy.polys.agca.ideals.Ideal method), 1022
reduce_element() (sympy.polys.agca.modules.SubModule method), 1020
reduced() (in module sympy.polys.polytools), 964
ReferenceFrame (class in sympy.physics.mechanics.essential), 1475
refine() (in module sympy.assumptions.refine), 1150
refine() (sympy.core.expr.Expr method), 114
refine_abs() (in module sympy.assumptions.refine), 1151
refine_exp() (in module sympy.assumptions.refine), 1151
refine_Pow() (in module sympy.assumptions.refine), 1150
refine_Relational() (in module sympy.assumptions.refine), 1151
refine_root() (in module sympy.polys.polytools), 962
refine_root() (sympy.polys.polyclasses.DMP method), 1047
refine_root() (sympy.polys.polytools.Poly method), 991
RefinementFailed (class in sympy.polys.polyerrors), 1112
reflect() (sympy.geometry.ellipse.Circle method), 476
reflect() (sympy.geometry.ellipse.Ellipse method), 473
reflect() (sympy.geometry.polygon.RegularPolygon method), 490
register_handler() (in module sympy.assumptions.ask), 1148
RegularPolygon (class in sympy.geometry.polygon), 484
Rel (class in sympy.core.relational), 134
rem() (in module sympy.polys.polytools), 953
rem() (sympy.polys.domains.domain.Domain method), 1034
rem() (sympy.polys.domains.field.Field method), 1035
rem() (sympy.polys.domains.ring.Ring method), 1036
rem() (sympy.polys.polyclasses.DMP method), 1047
rem() (sympy.polys.polytools.Poly method), 991
remove_handler() (in module sympy.assumptions.ask), 1148
removeO() (sympy.core.expr.Expr method), 114
render() (sympy.printing.pretty.stringpict.stringPict method), 1130
removeO() (sympy.polys.polytools.Poly method), 991
rep_expectation() (in module sympy.physics.quantum.represent), 1529
rep_innerproduct() (in module sympy.physics.quantum.represent), 1529
replace() (sympy.core.basic.Basic method), 92
replace() (sympy.matrices.matrices.MatrixBase method), 582
replace() (sympy.polys.polytools.Poly method), 992
Reporter (class in sympy.utilities.runtests), 1371
represent() (in module sympy.physics.quantum.represent), 1528
reprify() (sympy.printing.repr.ReprPrinter method), 1124
ReprPrinter (class in sympy.printing.repr), 1124
reset() (sympy.combinatorics.polyhedron.Polyhedron method), 229
reshape() (in module sympy.utilities.iterables), 1360
reshape() (sympy.matrices.sparse.SparseMatrix method), 605
residue() (in module sympy.series.residues), 1172
restrict_codomain() (sympy.polys.agca.homomorphisms.Mod method), 1030
restrict_domain() (sympy.polys.agca.homomorphisms.Modul method), 1030
Result (class in sympy.utilities.codegen), 1342
result_variables (sympy.utilities.codegen.Routine attribute), 1341
resultant() (in module sympy.polys.polytools), 955
resultant() (sympy.polys.polyclasses.DMP method), 1048
resultant() (sympy.polys.polytools.Poly method), 992
retract() (sympy.polys.polytools.Poly method), 992
revert() (sympy.polys.domains.domain.Domain method), 1034

revert() (sympy.polys.domains.field.Field method), 1035
revert() (sympy.polys.domains.ring.Ring method), 1036
revert() (sympy.polys.polyclasses.DMP method), 1048
revert() (sympy.polys.polytools.Poly method), 992
rewrite() (in module sympy.series.gruntz), 1167
rewrite() (sympy.core.basic.Basic method), 94
rf() (in module mpmath), 690
rgamma() (in module mpmath), 687
RGS (sympy.combinatorics.partitions.Partition attribute), 172
RGS_enum() (in module sympy.combinatorics.partitions), 176
RGS_generalized() (in module sympy.combinatorics.partitions), 175
RGS_rank() (in module sympy.combinatorics.partitions), 176
RGS_unrank() (in module sympy.combinatorics.partitions), 176
rhs() (sympy.physics.mechanics.kane.KanesMethod method), 1505
rhs() (sympy.physics.mechanics.lagrange.LagrangesMethod method), 1508
richardson() (in module mpmath), 893
richardson() (in module sympy.series.acceleration), 1171
Ridder (class in mpmath.calculus.optimization), 881
riemann_cyclic() (in module sympy.tensor.tensor), 1335
riemann_cyclic_replace() (in module sympy.tensor.tensor), 1335
riemannr() (in module mpmath), 868
right (sympy.core.sets.Interval attribute), 1176
right() (sympy.printing.pretty.stringpict.stringPict method), 1130
right_open (sympy.core.sets.Interval attribute), 1176
RigidBody (class in sympy.physics.mechanics.rigidbody), 1495
Ring (class in sympy.polys.domains.ring), 1035
RisingFactorial (class in sympy.functions.combinatorial.factorials), 333
Rk (in module sympy.physics.quantum.qft), 1557
RkGate (class in sympy.physics.quantum.qft), 1557
RL (sympy.matrices.sparse.SparseMatrix attribute), 601
rmul() (sympy.combinatorics.permutations.Permutation static method), 195
rmul_with_af() (sympy.combinatorics.permutations.Permuta static method), 196
root (in module sympy.printing.pretty.pretty_symbology), 1129
root() (in module mpmath), 659
root() (in module sympy.functions.elementary.miscellaneous), 317
root() (sympy.polys.polytools.Poly method), 993
root() (sympy.printing.pretty.stringpict.stringPict method), 1130
RootOf (class in sympy.polys.rootoftools), 1004
roots() (in module sympy.polys.polyroots), 1005
RootSum (class in sympy.polys.rootoftools), 1004
rot_axis1() (in module sympy.matrices.dense), 594
rot_axis2() (in module sympy.matrices.dense), 594
rot_axis3() (in module sympy.matrices.dense), 595
rotate() (sympy.combinatorics.polyhedron.Polyhedron method), 230
rotate() (sympy.geometry.curve.Curve method), 464
rotate() (sympy.geometry.ellipse.Ellipse method), 473
rotate() (sympy.geometry.entity.GeometryEntity method), 437
rotate() (sympy.geometry.point.Point method), 445
rotate() (sympy.geometry.polygon.RegularPolygon method), 490
rotate_left() (in module sympy.utilities.iterables), 1361
rotate_right() (in module sympy.utilities.iterables), 1361
Rotation (class in sympy.physics.quantum.spin), 1532

rotation (sympy.geometry.polygon.RegularPolygon attribute), 490
round() (sympy.core.expr.Expr method), 114
RoundFunction (class in sympy.functions.elementary.integers), 319
Routine (class in sympy.utilities.codegen), 1341
row() (sympy.matrices.sparse.SparseMatrix method), 605
row_del() (sympy.matrices.dense.MutableDenseMatrix method), 598
row_del() (sympy.matrices.sparse.MutableSparseMatrix method), 609
row_insert() (sympy.matrices.matrices.MatrixBase method), 582
row_join() (sympy.matrices.matrices.MatrixBase method), 583
row_join() (sympy.matrices.sparse.MutableSparseMatrix method), 610
row_list() (sympy.matrices.sparse.SparseMatrix method), 605
row_op() (sympy.matrices.dense.MutableDenseMatrix method), 598
row_op() (sympy.matrices.sparse.MutableSparseMatrix method), 610
row_structure_symbolic_cholesky() (sympy.matrices.sparse.SparseMatrix method), 605
row_swap() (sympy.matrices.dense.MutableDenseMatrix method), 598
row_swap() (sympy.matrices.sparse.MutableSparseMatrix method), 611
rref() (sympy.matrices.matrices.MatrixBase method), 583
rs_swap() (in module sympy.stats.rv), 1256
rsolve() (in module sympy.solvers.recurr), 1305
rsolve_hyper() (in module sympy.solvers.recurr), 1307
rsolve_poly() (in module sympy.solvers.recurr), 1306
rsolve_ratio() (in module sympy.solvers.recurr), 1306
run() (sympy.utilities.runtests.SymPyDocTestRunner method), 1372
run_all_tests() (in module sympy.utilities.runtests), 1373
run_in_subprocess_with_hash_randomization() (in module sympy.utilities.runtests), 1373
runs() (in module sympy.utilities.iterables), 1361
runs() (sympy.combinatorics.permutations.Permutation method), 196

S

S (in module sympy.core.singleton), 97
S (in module sympy.physics.quantum.gate), 1554
Sample (class in sympy.statistics.distributions), 1217
sample() (in module sympy.stats), 1254
sample_iter() (in module sympy.stats), 1254
satisfiable() (in module sympy.logic.inference), 547
saturate() (sympy.polys.agca.ideals.Ideal method), 1022
scalar_multiply() (sympy.matrices.sparse.SparseMatrix method), 606
scale() (sympy.geometry.curve.Curve method), 464
scale() (sympy.geometry.ellipse.Circle method), 477
scale() (sympy.geometry.ellipse.Ellipse method), 473
scale() (sympy.geometry.entity.GeometryEntity method), 437
scale() (sympy.geometry.point.Point method), 445
scale() (sympy.geometry.polygon.RegularPolygon method), 490
schreier_sims() (sympy.combinatorics.perm_groups.Permuta method), 223
schreier_sims_incremental() (sympy.combinatorics.perm_groups.PermutationGro method), 223
schreier_sims_random() (sympy.combinatorics.perm_groups.PermutationGro method), 224
schreier_vector() (sympy.combinatorics.perm_groups.Permu method), 225
scorergi() (in module mpmath), 756
scorerhi() (in module mpmath), 758
sdm_add() (in module sympy.polys.distributedmodules), 1096
sdm_deg() (in module sympy.polys.distributedmodules), 1098
sdm_ecart() (in module sympy.polys.distributedmodules), 1110
sdm_from_dict() (in module sympy.polys.distributedmodules), 1096

sdm_from_vector() (in module sympy.polys.distributedmodules), 1098
sdm_groebner() (in module sympy.polys.distributedmodules), 1111
sdm_LC() (in module sympy.polys.distributedmodules), 1096
sdm_LM() (in module sympy.polys.distributedmodules), 1097
sdm_LT() (in module sympy.polys.distributedmodules), 1097
sdm_monomial_deg() (in module sympy.polys.distributedmodules), 1095
sdm_monomial_divides() (in module sympy.polys.distributedmodules), 1095
sdm_monomial_mul() (in module sympy.polys.distributedmodules), 1095
sdm_mul_term() (in module sympy.polys.distributedmodules), 1097
sdm_nf_mora() (in module sympy.polys.distributedmodules), 1111
sdm_spoly() (in module sympy.polys.distributedmodules), 1110
sdm_to_dict() (in module sympy.polys.distributedmodules), 1096
sdm_to_vector() (in module sympy.polys.distributedmodules), 1098
sdm_zero() (in module sympy.polys.distributedmodules), 1098
search() (sympy.ntheory.generate.Sieve method), 262
search_function (sympy.physics.quantum.grover.OracleGate attribute), 1555
sec() (in module mpmath), 675
Secant (class in mpmath.calculus.optimization), 879
sech() (in module mpmath), 682
secondzeta() (in module mpmath), 856
Segment (class in sympy.geometry.line), 458
select() (sympy.simplify.epathtools.EPath method), 1203
selections (sympy.combinatorics.graycode.GrayCode attribute), 243
separate() (in module sympy.simplify.simplify), 1187
separate() (sympy.core.expr.Expr method), 115
separatevars() (in module sympy.simplify.simplify), 1187
series() (in module sympy.series.series), 1169
series() (sympy.core.expr.Expr method), 115
Set (class in sympy.core.sets), 1172
set_acc() (sympy.physics.mechanics.point.Point method), 1489
set_ang_acc() (sympy.physics.mechanics.essential.Reference method), 1478
set_ang_vel() (sympy.physics.mechanics.essential.Reference method), 1479
set_comm() (sympy.tensor.tensor._TensorManager method), 1325
set_comms() (sympy.tensor.tensor._TensorManager method), 1326
set_domain() (sympy.polys.polytools.Poly method), 993
set_global_settings() (sympy.printing.printer.Printer class method), 1116
set_modulus() (sympy.polys.polytools.Poly method), 993
set_pos() (sympy.physics.mechanics.point.Point method), 1489
set_potential_energy() (sympy.physics.mechanics.particle.Particle method), 1495
set_potential_energy() (sympy.physics.mechanics.rigidbody.RigidBody method), 1498
set_vel() (sympy.physics.mechanics.point.Point method), 1490
seterr() (in module sympy.core.numbers), 125
shanks() (in module mpmath), 894
shanks() (in module sympy.series.acceleration), 1171
shape (sympy.matrices.matrices.MatrixBase attribute), 583
shape (sympy.tensor.indexed.Indexed attribute), 1320
shape (sympy.tensor.indexed.IndexedBase attribute), 1321
ShapeError (class in sympy.matrices.matrices), 588
Shi (class in sympy.functions.special.error_functions), 349
shi() (in module mpmath), 709

shift() (sympy.polys.polyclasses.DMP method), 1048
shift() (sympy.polys.polytools.Poly method), 993
shor() (in module sympy.physics.quantum.shor), 1562
Si (class in sympy.functions.special.error_functions), 347
si() (in module mpmath), 708
sides (sympy.geometry.polygon.Polygon attribute), 483
siegeltheta() (in module mpmath), 846
siegelz() (in module mpmath), 844
Sieve (class in sympy.ntheory.generate), 261
sift() (in module sympy.utilities.iterables), 1361
sign (class in sympy.functions.elementary.complexes), 321
sign() (in module mpmath), 640
sign() (in module sympy.series.gruntz), 1167
signature() (sympy.combinatorics.permutations.Permutation method), 196
SimpleDomain (class in sympy.polys.domains.simpledomain), 1036
simplify() (in module sympy.simplify.simplify), 1182
simplify() (sympy.core.expr.Expr method), 116
simplify() (sympy.functions.special.delta_functions.DiracDelta method), 337
simplify() (sympy.matrices.dense.MutableDenseMatrix method), 599
simplify() (sympy.matrices.matrices.MatrixBase method), 583
simplify() (sympy.physics.mechanics.essential.Dyadic method), 1484
simplify() (sympy.physics.mechanics.essential.Vector method), 1482
simplify_index_permutations() (in module sympy.physics.secondquant), 1417
simplify_logic() (in module sympy.logic.boolalg), 546
sin (class in sympy.functions.elementary.trigonometric), 319
sin() (in module mpmath), 674
sinc() (in module mpmath), 679
sincpi() (in module mpmath), 680
sine_transform() (in module sympy.integrals.transforms), 502
SingleDomain (class in sympy.stats.rv), 1255
SinglePSpace (class in sympy.stats.rv), 1255
singular_values() (sympy.matrices.matrices.MatrixBase method), 584
sinh (class in sympy.functions.elementary.hyperbolic), 319
sinh() (in module mpmath), 681
sinm() (in module mpmath), 923
sinpi() (in module mpmath), 676
size (sympy.combinatorics.permutations.Permutation attribute), 196
size (sympy.combinatorics.polyhedron.Polyhedron attribute), 231
size (sympy.combinatorics.prufer.Prufer attribute), 233
size (sympy.combinatorics.subsets.Subset attribute), 239
SKIP() (in module sympy.utilities.pytest), 1369
skip() (sympy.combinatorics.graycode.GrayCode method), 243
slice() (sympy.polys.polyclasses.DMP method), 1048
slice() (sympy.polys.polytools.Poly method), 993
slope (sympy.geometry.line.LinearEntity attribute), 454
smoothness() (in module sympy.ntheory.factor_), 266
smoothness_p() (in module sympy.ntheory.factor_), 267
solve() (in module sympy.solvers.solvers), 1295
solve() (sympy.matrices.matrices.MatrixBase method), 584
solve() (sympy.matrices.sparse.SparseMatrix method), 606
solve_congruence() (in module sympy.ntheory.modular), 277
solve_least_squares() (sympy.matrices.matrices.MatrixBase method), 584
solve_least_squares() (sympy.matrices.sparse.SparseMatrix method), 606
solve_linear() (in module sympy.solvers.solvers), 1300
solve_linear_system() (in module sympy.solvers.solvers), 1301
solve_linear_system_LU() (in module sympy.solvers.solvers), 1302
solve_poly_system() (in module sympy.solvers.polysys), 1308
solve_triangulated() (in module sympy.solvers.polysys), 1308
solve_undetermined_coeffs() (in module sympy.solvers.solvers), 1302

SOPform() (in module sympy.logic.boolalg), 542
sort_key() (sympy.combinatorics.partitions.Partition method), 173
sort_key() (sympy.core.basic.Basic method), 94
sort_key() (sympy.core.sets.Set method), 1173
sorted_components() (sympy.tensor.tensor.TensMul method), 1334
source (sympy.geometry.line.Ray attribute), 457
source() (in module sympy.utilities.source), 1378
SparseMatrix (class in sympy.matrices.sparse), 601
spherharm() (in module mpmath), 787
spin() (sympy.geometry.polygon.RegularPolygon method), 490
split() (sympy.tensor.tensor.TensMul method), 1334
split_super_sub() (in module sympy.printing.conventions), 1127
split_symbols() (in module sympy.parsing.sympy_parser), 1380
split_symbols_custom() (in module sympy.parsing.sympy_parser), 1380
splot() (in module mpmath), 655
spoly() (in module sympy.polys.groebnertools), 1109
sqf() (in module sympy.polys.polytools), 960
sqf_list() (in module sympy.polys.polytools), 960
sqf_list() (sympy.polys.polyclasses.DMP method), 1048
sqf_list() (sympy.polys.polytools.Poly method), 993
sqf_list_include() (sympy.polys.polyclasses.DMP method), 1048
sqf_list_include() (sympy.polys.polytools.Poly method), 994
sqf_norm() (in module sympy.polys.polytools), 960
sqf_norm() (sympy.polys.polyclasses.DMP method), 1048
sqf_norm() (sympy.polys.polytools.Poly method), 994
sqf_part() (in module sympy.polys.polytools), 960
sqf_part() (sympy.polys.polyclasses.DMP method), 1048
sqf_part() (sympy.polys.polytools.Poly method), 995
sqr() (sympy.polys.polyclasses.DMP method), 1048
sqr() (sympy.polys.polytools.Poly method), 995
sqrt (class in sympy.functions.elementary.miscellaneous), 320
sqrt() (in module mpmath), 658
sqrt() (sympy.polys.domains.domain.Domain method), 1034
sqrtdenest() (in module sympy.simplify.sqrtdenest), 1200
sqrtm() (in module mpmath), 924
srepr() (in module sympy.printing.repr), 1124
sstrrepr() (in module sympy.printing.str), 1125
stabilizer() (sympy.combinatorics.perm_groups.Permutation method), 226
stack() (sympy.printing.pretty.stringpict.stringPict static method), 1130
standard_transformations (in module sympy.parsing.sympy_parser), 1380
start (sympy.core.sets.Interval attribute), 1176
State (class in sympy.physics.quantum.state), 1542
state() (sympy.physics.secondquant.FixedBosonicBasis method), 1409
state() (sympy.physics.secondquant.VarBosonicBasis method), 1408
state_to_operators() (in module sympy.physics.quantum.operatorset), 1526
StateBase (class in sympy.physics.quantum.state), 1542
std() (in module sympy.stats), 1254
stieltjes() (in module mpmath), 842
stirling() (in module sympy.functions.combinatorial.numbers), 334
StopTokenizing (class in sympy.parsing.sympy_tokenize), 1380
StrictGreaterThan (class in sympy.core.relational), 142
StrictLessThan (class in sympy.core.relational), 145
stringify_expr() (in module sympy.parsing.sympy_parser), 1379
stringPict (class in sympy.printing.pretty.stringpict), 1129
strong_gens (sympy.combinatorics.perm_groups.Permutatio attribute), 226
StrPrinter (class in sympy.printing.str), 1124

struveh() (in module mpmath), 737
struvel() (in module mpmath), 738
StudentT() (in module sympy.stats), 1245
sturm() (in module sympy.polys.polytools), 959
sturm() (sympy.polys.polyclasses.DMP method), 1048
sturm() (sympy.polys.polytools.Poly method), 995
sub (in module sympy.printing.pretty.pretty_symbology), 1128
sub() (sympy.polys.domains.domain.Domain method), 1034
sub() (sympy.polys.polyclasses.DMF method), 1049
sub() (sympy.polys.polyclasses.DMP method), 1048
sub() (sympy.polys.polytools.Poly method), 995
sub_ground() (sympy.polys.polyclasses.DMP method), 1048
sub_ground() (sympy.polys.polytools.Poly method), 996
subdiagram_from_objects() (sympy.categories.Diagram method), 1571
subgroup_search() (sympy.combinatorics.perm_groups.PermutationGroup method), 226
submatrix() (sympy.matrices.matrices.MatrixBase method), 585
SubModule (class in sympy.polys.agca.modules), 1017
submodule() (sympy.polys.agca.modules.Module method), 1015
submodule() (sympy.polys.agca.modules.QuotientModule method), 1024
submodule() (sympy.polys.agca.modules.SubModule method), 1020
SubQuotientModule (class in sympy.polys.agca.modules), 1024
subresultants() (in module sympy.polys.polytools), 955
subresultants() (sympy.polys.polyclasses.DMP method), 1048
subresultants() (sympy.polys.polytools.Poly method), 996
Subs (class in sympy.core.function), 156
subs() (sympy.core.basic.Basic method), 94
subs() (sympy.matrices.immutable.ImmutableSparseMatrix method), 612
subs() (sympy.matrices.matrices.MatrixBase method), 585
subs() (sympy.physics.mechanics.essential.Dyadic method), 1484
subs() (sympy.physics.mechanics.essential.Vector method), 1482
Subset (class in sympy.combinatorics.subsets), 235
subset (sympy.combinatorics.subsets.Subset attribute), 239
subset() (sympy.core.sets.Set method), 1174
subset() (sympy.polys.agca.ideals.Ideal method), 1022
subset() (sympy.polys.agca.modules.Module method), 1015
subset_from_bitlist() (sympy.combinatorics.subsets.Subset class method), 239
subset_indices() (sympy.combinatorics.subsets.Subset class method), 240
subsets() (in module sympy.utilities.iterables), 1362
SubsSet (class in sympy.series.gruntz), 1168
substitute_dummies() (in module sympy.physics.secondquant), 1416
substitute_indices() (sympy.tensor.tensor.TensAdd method), 1331
substitute_indices() (sympy.tensor.tensor.TensMul method), 1334
Sum (class in sympy.concrete.summations), 283
sum_next() (mpmath.calculus.quadrature.QuadratureRule method), 907
sum_next() (mpmath.calculus.quadrature.TanhSinh method), 908
sumap() (in module mpmath), 889
sumem() (in module mpmath), 888
summation() (in module sympy.concrete.summations), 290
summation() (mpmath.calculus.quadrature.QuadratureRule method), 907
sup (in module sympy.printing.pretty.pretty_symbology), 1128
sup (sympy.core.sets.Set attribute), 1174
superfac() (in module mpmath), 693
superposition_basis() (in module sympy.physics.quantum.grover), 1555
superset (sympy.combinatorics.subsets.Subset attribute), 240
superset_size (sympy.combinatorics.subsets.Subset attribute), 240
support() (sympy.combinatorics.permutations.Permutation method), 197
SurfaceBaseSeries (class in sympy.plotting.plot), 1144
SurfaceOver2DRangeSeries (class in sympy.plotting.plot), 1144

SWAP (in module sympy.physics.quantum.gate), 1553
swap_point() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1550
SwapGate (class in sympy.physics.quantum.gate), 1553
swinnerton_dyer_poly() (in module sympy.polys.specialpolys), 1005
symarray() (in module sympy.matrices.dense), 593
symb_2txt (in module sympy.printing.pretty.pretty_symbology), 1128
Symbol (class in sympy.core.symbol), 117
symbol (sympy.physics.secondquant.AntiSymmetricTensor attribute), 1416
Symbol() (sympy.assumptions.handlers.calculus.AskBoundedHandler static method), 1153
symbols() (in module sympy.core.symbol), 118
symmetric() (sympy.combinatorics.generators static method), 200
symmetric_poly() (in module sympy.polys.specialpolys), 1005
symmetric_residue() (in module sympy.ntheory.modular), 275
SymmetricGroup() (in module sympy.combinatorics.named_groups), 246
symmetrize() (in module sympy.polys.polyfuncs), 1001
sympify() (in module sympy.core.sympify), 81
sympy (module), 612
sympy.assumptions (module), 1147
sympy.assumptions.ask (module), 1147
sympy.assumptions.assume (module), 1149
sympy.assumptions.handlers (module), 1151
sympy.assumptions.handlers.calculus (module), 1151
sympy.assumptions.handlers.ntheory (module), 1153
sympy.assumptions.handlers.order (module), 1154
sympy.assumptions.handlers.sets (module), 1154
sympy.assumptions.refine (module), 1150
sympy.categories (module), 1563
sympy.categories.diagram_drawing (module), 1571
sympy.combinatorics.generators (module), 200
sympy.combinatorics.graycode (module), 241
sympy.combinatorics.group_constructs (module), 254
sympy.combinatorics.named_groups (module), 246
sympy.combinatorics.partitions (module), 172
sympy.combinatorics.perm_groups (module), 201
sympy.combinatorics.permutations (module), 177
sympy.combinatorics.polyhedron (module), 228
sympy.combinatorics.prufer (module), 231
sympy.combinatorics.subsets (module), 235
sympy.combinatorics.tensor_can (module), 255
sympy.combinatorics.testutil (module), 254
sympy.combinatorics.util (module), 248
sympy.core.add (module), 131
sympy.core.basic (module), 84
sympy.core.cache (module), 84
sympy.core.compatibility (module), 168
sympy.core.containers (module), 167
sympy.core.evalf (module), 167
sympy.core.expr (module), 97
sympy.core.exprtools (module), 170
sympy.core.function (module), 149
sympy.core.mod (module), 133
sympy.core.mul (module), 128
sympy.core.multidimensional (module), 148
sympy.core.numbers (module), 120
sympy.core.power (module), 127
sympy.core.relational (module), 134
sympy.core.sets (module), 1172
sympy.core.singleton (module), 97
sympy.core.symbol (module), 117
sympy.core.sympify (module), 81
sympy.crypto.crypto (module), 282
sympy.diffgeom (module), 1580
sympy.functions (module), 302
sympy.functions.special.elliptic_integrals (module), 378
sympy.functions.special.error_functions (module), 341
sympy.functions.special.polynomials (module), 381
sympy.functions.special.zeta_functions (module), 368
sympy.galgebra (module), 398
sympy.galgebra.debug (module), 432
sympy.galgebra.ga (module), 399
sympy.galgebra.ga.Com() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.cross() (in module sympy.galgebra.ga), 416

sympy.galgebra.ga.DD() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.dual() (in module sympy.galgebra.ga), 416
sympy.galgebra.ga.Fmt() (in module sympy.galgebra.ga), 419
sympy.galgebra.ga.Format() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.ga_print_off() (in module sympy.galgebra.ga), 419
sympy.galgebra.ga.ga_print_on() (in module sympy.galgebra.ga), 419
sympy.galgebra.ga.inv() (in module sympy.galgebra.ga), 416
sympy.galgebra.ga.MV (class in sympy.galgebra.ga), 413
sympy.galgebra.ga.MV.convert_from_blades() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.convert_to_blades() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.dd() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.diff() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.dual() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.even() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.exp() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.expand() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.factor() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.func() (in module sympy.galgebra.ga), 414
sympy.galgebra.ga.MV.grade() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.inv() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.norm() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.rev() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.scalar() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.set_coef() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.setup() (in module sympy.galgebra.ga), 411
sympy.galgebra.ga.MV.simplify() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.subs() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.MV.trigsimp() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.Nga() (in module sympy.galgebra.ga), 415
sympy.galgebra.ga.proj() (in module sympy.galgebra.ga), 416
sympy.galgebra.ga.ReciprocalFrame() (in module sympy.galgebra.ga), 416
sympy.galgebra.ga.refl() (in module sympy.galgebra.ga), 416
sympy.galgebra.ga.rot() (in module sympy.galgebra.ga), 416
sympy.galgebra.ga.ScalarFunction() (in module sympy.galgebra.ga), 416
sympy.galgebra.manifold (module), 432
sympy.galgebra.ncutil (module), 432
sympy.galgebra.precedence (module), 432
sympy.galgebra.precedence.define_precedence() (in module sympy.galgebra.ga), 402, 416
sympy.galgebra.precedence.GAeval() (in module sympy.galgebra.ga), 402, 415
sympy.galgebra.printing (module), 432
sympy.galgebra.printing.Format() (in module sympy.galgebra.ga), 419
sympy.galgebra.printing.GA_Printer() (in module sympy.galgebra.ga), 419
sympy.galgebra.printing.Get_Program() (in module sympy.galgebra.ga), 422
sympy.galgebra.printing.Print_Function() (in module sympy.galgebra.ga), 422
sympy.galgebra.printing.xdvi() (in module sympy.galgebra.ga), 419
sympy.galgebra.stringarrays (module), 432
sympy.galgebra.vector (module), 432
sympy.geometry (module), 432
sympy.geometry.curve (module), 461
sympy.geometry.ellipse (module), 464
sympy.geometry.entity (module), 436
sympy.geometry.line (module), 446
sympy.geometry.point (module), 440
sympy.geometry.polygon (module), 477
sympy.geometry.util (module), 438
sympy.integrals (module), 498
sympy.integrals.meijerint_doc (module), 519
sympy.integrals.transforms (module), 499
sympy.logic (module), 541
sympy.matrices (module), 547
sympy.matrices.expressions (module), 614
sympy.matrices.expressions.blockmatrix (module), 618
sympy.matrices.immutable (module), 612
sympy.matrices.matrices (module), 547

sympy.matrices.sparse (module), 601
sympy.ntheory.factor_ (module), 266
sympy.ntheory.generate (module), 261
sympy.ntheory.modular (module), 275
sympy.ntheory.multinomial (module), 277
sympy.ntheory.partitions_ (module), 279
sympy.ntheory.primetest (module), 279
sympy.ntheory.residue_ntheory (module), 280
sympy.physics (module), 1382
sympy.physics.gaussopt (module), 1382
sympy.physics.hep.gamma_matrices (module), 1426
sympy.physics.hydrogen (module), 1392
sympy.physics.matrices (module), 1395
sympy.physics.mechanics.functions (module), 1492, 1505
sympy.physics.mechanics.kane (module), 1503
sympy.physics.mechanics.lagrange (module), 1506
sympy.physics.mechanics.particle (module), 1492
sympy.physics.mechanics.point (module), 1487
sympy.physics.mechanics.rigidbody (module), 1495
sympy.physics.paulialgebra (module), 1396
sympy.physics.qho_1d (module), 1396
sympy.physics.quantum.anticommutator (module), 1510
sympy.physics.quantum.cartesian (module), 1518
sympy.physics.quantum.cg (module), 1511
sympy.physics.quantum.circuitplot (module), 1549
sympy.physics.quantum.commutator (module), 1513
sympy.physics.quantum.constants (module), 1514
sympy.physics.quantum.dagger (module), 1514
sympy.physics.quantum.gate (module), 1550
sympy.physics.quantum.grover (module), 1555
sympy.physics.quantum.hilbert (module), 1519
sympy.physics.quantum.innerproduct (module), 1515
sympy.physics.quantum.operator (module), 1521
sympy.physics.quantum.operatorset (module), 1525
sympy.physics.quantum.piab (module), 1562
sympy.physics.quantum.qapply (module), 1527
sympy.physics.quantum.qft (module), 1557
sympy.physics.quantum.qubit (module), 1557
sympy.physics.quantum.represent (module), 1528
sympy.physics.quantum.shor (module), 1561
sympy.physics.quantum.spin (module), 1532
sympy.physics.quantum.state (module), 1542
sympy.physics.quantum.tensorproduct (module), 1516
sympy.physics.secondquant (module), 1398
sympy.physics.sho (module), 1397
sympy.physics.units (module), 1424
sympy.physics.wigner (module), 1418
sympy.plotting.plot (module), 1132
sympy.plotting.pygletplot (module), 1144
sympy.printing.ccode (module), 1117
sympy.printing.codeprinter (module), 1127
sympy.printing.conventions (module), 1127
sympy.printing.fcode (module), 1118
sympy.printing.gtk (module), 1121
sympy.printing.lambdarepr (module), 1121
sympy.printing.latex (module), 1121
sympy.printing.mathml (module), 1123
sympy.printing.precedence (module), 1128
sympy.printing.pretty.pretty (module), 1116
sympy.printing.pretty.pretty_symbology (module), 1128
sympy.printing.pretty.stringpict (module), 1129
sympy.printing.preview (module), 1126
sympy.printing.printer (module), 1113
sympy.printing.python (module), 1124
sympy.printing.repr (module), 1124
sympy.printing.str (module), 1124
sympy.printing.tree (module), 1125
sympy.series (module), 1164
sympy.sets.fancysets (module), 1181
sympy.simplify.cse_main (module), 1163, 1200
sympy.simplify.epathtools (module), 1202
sympy.simplify.hyperexpand (module), 1201
sympy.simplify.hyperexpand_doc (module), 1212
sympy.simplify.simplify (module), 68
sympy.simplify.sqrtdenest (module), 1200
sympy.simplify.traversaltools (module), 1201
sympy.solvers (module), 1295
sympy.solvers.ode (module), 1284
sympy.solvers.pde (module), 1294
sympy.solvers.recurr (module), 1305
sympy.statistics (module), 1215
sympy.stats (module), 1222

sympy.stats.crv (module), 1255
sympy.stats.crv_types (module), 1256
sympy.stats.Die() (in module sympy.stats.crv_types), 1256
sympy.stats.frv (module), 1255
sympy.stats.frv_types (module), 1256
sympy.stats.Normal() (in module sympy.stats.crv_types), 1256
sympy.stats.rv (module), 1255
sympy.tensor (module), 1315
sympy.tensor.index_methods (module), 1322
sympy.tensor.indexed (module), 1315
sympy.tensor.tensor (module), 1325
sympy.utilities (module), 1335
sympy.utilities.autowrap (module), 1336
sympy.utilities.codegen (module), 1340
sympy.utilities.decorator (module), 1345
sympy.utilities.iterables (module), 1348
sympy.utilities.lambdify (module), 1365
sympy.utilities.memoization (module), 1367
sympy.utilities.misc (module), 1367
sympy.utilities.pkgdata (module), 1368
sympy.utilities.pytest (module), 1369
sympy.utilities.randtest (module), 1370
sympy.utilities.runtests (module), 1371
sympy.utilities.source (module), 1377
sympy.utilities.timeutils (module), 1378
SymPyDocTestFinder (class in sympy.utilities.runtests), 1371
SymPyDocTestRunner (class in sympy.utilities.runtests), 1372
SymPyOutputChecker (class in sympy.utilities.runtests), 1372
sympytestfile() (in module sympy.utilities.runtests), 1374
SymPyTestResults (in module sympy.utilities.runtests), 1372
syzygy_module() (sympy.polys.agca.modules.SubModuleOpt method), 1020

T

T (in module sympy.physics.quantum.gate), 1554
T (sympy.matrices.expressions.MatrixExpr attribute), 614
T (sympy.matrices.matrices.MatrixBase attribute), 560
t (sympy.physics.quantum.shor.CMod attribute), 1562
table() (sympy.matrices.matrices.MatrixBase method), 586
take() (in module sympy.utilities.iterables), 1363
tan (class in sympy.functions.elementary.trigonometric), 321
tan() (in module mpmath), 674
tangent_lines() (sympy.geometry.ellipse.Ellipse method), 474
tanh (class in sympy.functions.elementary.hyperbolic), 322
tanh() (in module mpmath), 681
TanhSinh (class in mpmath.calculus.quadrature), 907
targets (sympy.physics.quantum.gate.CGate attribute), 1551
targets (sympy.physics.quantum.gate.CNotGate attribute), 1553
targets (sympy.physics.quantum.gate.Gate attribute), 1551
targets (sympy.physics.quantum.gate.UGate attribute), 1551
targets (sympy.physics.quantum.grover.OracleGate attribute), 1555
taufrom() (in module mpmath), 816
taylor() (in module mpmath), 911
taylor_term() (sympy.core.function.Function class method), 155
taylor_term() (sympy.functions.elementary.exponential.exp static method), 310
taylor_term() (sympy.functions.elementary.exponential.log static method), 313
taylor_term() (sympy.functions.elementary.hyperbolic.sinh static method), 320
TC() (sympy.polys.polyclasses.ANP method), 1050
TC() (sympy.polys.polyclasses.DMP method), 1044
TC() (sympy.polys.polytools.Poly method), 967
TensAdd (class in sympy.tensor.tensor), 1330
TensExpr (class in sympy.tensor.tensor), 1329
TensMul (class in sympy.tensor.tensor), 1331
tensor_indices() (in module sympy.tensor.tensor), 1328
tensor_mul() (in module sympy.tensor.tensor), 1335
tensor_product_simp() (in module sympy.physics.quantum.tensorproduct), 1517
TensorHead (class in sympy.tensor.tensor), 1329
TensorIndex (class in sympy.tensor.tensor), 1327
TensorIndexType (class in sympy.tensor.tensor), 1326
TensorProduct (class in sympy.physics.quantum.tensorproduct),

1516
TensorSymmetry (class in sympy.tensor.tensor), 1328
TensorType (class in sympy.tensor.tensor), 1328
terminal_width() (sympy.printing.pretty.stringpict.stringPict method), 1130
terms() (sympy.polys.polyclasses.DMP method), 1048
terms() (sympy.polys.polytools.Poly method), 996
terms_gcd() (in module sympy.polys.polytools), 955
terms_gcd() (sympy.polys.polyclasses.DMP method), 1048
terms_gcd() (sympy.polys.polytools.Poly method), 996
termwise() (sympy.polys.polytools.Poly method), 997
test() (in module sympy.utilities.runtests), 1376
test_derivative_numerically() (in module sympy.utilities.randtest), 1370
test_numerically() (in module sympy.utilities.randtest), 1370
TGate (class in sympy.physics.quantum.gate), 1552
ThinLens (class in sympy.physics.gaussopt), 1389
threaded() (in module sympy.utilities.decorator), 1346
threaded_factory() (in module sympy.utilities.decorator), 1346
time (sympy.physics.quantum.state.TimeDepState attribute), 1544
timed() (in module sympy.utilities.timeutils), 1378
TimeDepBra (class in sympy.physics.quantum.state), 1544
TimeDepKet (class in sympy.physics.quantum.state), 1545
TimeDepState (class in sympy.physics.quantum.state), 1544
timing() (in module mpmath), 652
to_algebraic_integer() (sympy.polys.numberfields.AlgebraicNumber method), 1004
to_cnf() (in module sympy.logic.boolalg), 545
to_dict() (sympy.polys.polyclasses.ANP method), 1050
to_dict() (sympy.polys.polyclasses.DMP method), 1048
to_dnf() (in module sympy.logic.boolalg), 545
to_exact() (sympy.polys.polyclasses.DMP method), 1048
to_exact() (sympy.polys.polytools.Poly method), 997
to_field() (sympy.polys.polyclasses.DMP method), 1048
to_field() (sympy.polys.polytools.Poly method), 997
to_list() (sympy.polys.polyclasses.ANP method), 1050
to_number_field() (in module sympy.polys.numberfields), 1004
to_prufer() (sympy.combinatorics.prufer.Prufer static method), 234
to_ring() (sympy.polys.polyclasses.DMP method), 1048
to_ring() (sympy.polys.polytools.Poly method), 997
to_sympy() (sympy.polys.domains.AlgebraicField method), 1040
to_sympy() (sympy.polys.domains.domain.Domain method), 1034
to_sympy() (sympy.polys.domains.ExpressionDomain method), 1043
to_sympy() (sympy.polys.domains.FiniteField method), 1038
to_sympy() (sympy.polys.domains.FractionField method), 1042
to_sympy_dict() (sympy.polys.polyclasses.ANP method), 1050
to_sympy_dict() (sympy.polys.polyclasses.DMP method), 1048
to_sympy_list() (sympy.polys.polyclasses.ANP method), 1050
to_tree() (sympy.combinatorics.prufer.Prufer static method), 234
to_tuple() (sympy.polys.polyclasses.ANP method), 1050
to_tuple() (sympy.polys.polyclasses.DMP method), 1048
together() (in module sympy.polys.rationaltools), 1006
together() (sympy.core.expr.Expr method), 116
TokenError (class in sympy.parsing.sympy_tokenize), 1380
tokenize() (in module sympy.parsing.sympy_tokenize), 1379
tolist() (sympy.matrices.sparse.SparseMatrix method), 607
topological_sort() (in module sympy.utilities.iterables), 1363

total_degree() (sympy.polys.polyclasses.DMP method), 1048
total_degree() (sympy.polys.polytools.Poly method), 998
totient() (in module sympy.ntheory.factor_), 275
Trace (class in sympy.matrices.expressions), 617
trailing() (in module sympy.ntheory.factor_), 268
transform() (sympy.geometry.point.Point method), 445
transform() (sympy.integrals.Integral method), 533
transform() (sympy.statistics.distributions.PDF method), 1222
transform_nodes() (mpmath.calculus.quadrature.QuadratureRule method), 907
transitivity_degree (sympy.combinatorics.perm_groups.PermutationGroup attribute), 228
translate() (sympy.geometry.curve.Curve method), 464
translate() (sympy.geometry.entity.GeometryEntity method), 437
translate() (sympy.geometry.point.Point method), 445
Transpose (class in sympy.matrices.expressions), 617
transpose() (sympy.matrices.expressions.blockmatrix.BlockMatrix method), 619
transpositions() (sympy.combinatorics.permutations.Permutation method), 197
tree() (in module sympy.printing.tree), 1125
tree_repr (sympy.combinatorics.prufer.Prufer attribute), 234
Triangle (class in sympy.geometry.polygon), 491
Triangular() (in module sympy.stats), 1246
trigamma() (in module sympy.functions.special.gamma_functions), 339
trigintegrate() (sympy.integrals static method), 528
trigsimp() (in module sympy.simplify.simplify), 1191
trigsimp() (sympy.core.expr.Expr method), 116
trunc() (in module sympy.polys.polytools), 957
trunc() (sympy.polys.polyclasses.DMP method), 1049
trunc() (sympy.polys.polytools.Poly method), 998
tsolve() (in module sympy.solvers.solvers), 1303
Tuple (class in sympy.core.containers), 167
twinprime (mpmath.mp attribute), 657
TwoQubitGate (class in sympy.physics.quantum.gate), 1551

U

U() (in module sympy.printing.pretty.pretty_symbology), 1128
ufuncify() (in module sympy.utilities.autowrap), 1338
UGate (class in sympy.physics.quantum.gate), 1551
uncouple() (in module sympy.physics.quantum.spin), 1541
Unequality (class in sympy.core.relational), 142
unflatten() (in module sympy.utilities.iterables), 1364
UnificationFailed (class in sympy.polys.polyerrors), 1112
Uniform (class in sympy.statistics.distributions), 1220
Uniform() (in module sympy.stats), 1247
UniformSum() (in module sympy.stats), 1248
unify() (sympy.polys.domains.domain.Domain method), 1034
unify() (sympy.polys.polyclasses.ANP method), 1050
unify() (sympy.polys.polyclasses.DMP method), 1049
unify() (sympy.polys.polytools.Poly method), 998
Union (class in sympy.core.sets), 1177
union() (sympy.core.sets.Set method), 1174
union() (sympy.polys.agca.ideals.Ideal method), 1023
union() (sympy.polys.agca.modules.SubModule method), 1020
union() (sympy.series.gruntz.SubsSet method), 1168
uniq() (in module sympy.utilities.iterables), 1364
Unit (class in sympy.physics.units), 1426
unit (sympy.polys.polytools.Poly attribute), 999
UnitaryOperator (class in sympy.physics.quantum.operator), 1522
unitroots() (in module mpmath), 660
UnivariatePolynomialError (class in sympy.polys.polyerrors), 1112
UniversalSet (class in sympy.core.sets), 1180

unrank() (sympy.combinatorics.graycode.GrayCode class method), 244
unrank() (sympy.combinatorics.prufer.Prufer class method), 235
unrank_binary() (sympy.combinatorics.subsets.Subset class method), 240
unrank_gray() (sympy.combinatorics.subsets.Subset class method), 241
unrank_lex() (sympy.combinatorics.permutations.Permutation class method), 197
unrank_nonlex() (sympy.combinatorics.permutations.Permutation class method), 198
unrank_trotterjohnson() (sympy.combinatorics.permutations.Permutation class method), 198
unrestricted_necklace() (in module sympy.utilities.iterables), 1364
untokenize() (in module sympy.parsing.sympy_tokenize), 1379
update() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1550
upper (sympy.physics.secondquant.AntiSymmetricTensor attribute), 1416
upper (sympy.tensor.indexed.Idx attribute), 1318
upper_triangular_solve() (sympy.matrices.matrices.MatrixBase method), 586
uppergamma (class in sympy.functions.special.gamma_functions), 339
use() (in module sympy.simplify.traversaltools), 1201

V

v1pt_theory() (sympy.physics.mechanics.point.Point method), 1490
v2pt_theory() (sympy.physics.mechanics.point.Point method), 1491
values() (sympy.core.containers.Dict method), 168
values() (sympy.matrices.matrices.MatrixBase method), 586
var() (in module sympy.core.symbol), 120
VarBosonicBasis (class in sympy.physics.secondquant), 1408
variables (sympy.concrete.products.Product attribute), 290
variables (sympy.concrete.summations.Sum attribute), 286
variables (sympy.core.function.Lambda attribute), 149
variables (sympy.core.function.Subs attribute), 156
variables (sympy.integrals.Integral attribute), 534
variables (sympy.physics.quantum.operator.DifferentialOperator attribute), 1525
variables (sympy.physics.quantum.state.Wavefunction attribute), 1549
variables (sympy.utilities.codegen.Routine attribute), 1341
variance() (in module sympy.stats), 1253
variations() (in module sympy.utilities.iterables), 1364
vec() (sympy.matrices.matrices.MatrixBase method), 587
vech() (sympy.matrices.matrices.MatrixBase method), 587
Vector (class in sympy.physics.mechanics.essential), 1480
vectorize (class in sympy.core.multidimensional), 148
vel() (sympy.physics.mechanics.point.Point method), 1491
vertices (sympy.combinatorics.polyhedron.Polyhedron attribute), 231
vertices (sympy.geometry.polygon.Polygon attribute), 483
vertices (sympy.geometry.polygon.RegularPolygon attribute), 491
vertices (sympy.geometry.polygon.Triangle attribute), 497
VF() (in module sympy.printing.pretty.pretty_symbology), 1129
viete() (in module sympy.polys.polyfuncs), 1002
vobj() (in module sympy.printing.pretty.pretty_symbology), 1129
VonMises() (in module sympy.stats), 1248
vradius (sympy.geometry.ellipse.Circle attribute), 477
vradius (sympy.geometry.ellipse.Ellipse attribute), 474
vstack() (sympy.matrices.matrices.MatrixBase class method), 587

W

w (sympy.physics.gaussopt.BeamParameter attribute), 1384
w_0 (sympy.physics.gaussopt.BeamParameter attribute), 1384
waist2rayleigh() (in module sympy.physics.gaussopt), 1392

waist_approximation_limit (sympy.physics.gaussopt.BeamParameter attribute), 1385
Wavefunction (class in sympy.physics.quantum.state), 1545
webere() (in module mpmath), 740
Weibull() (in module sympy.stats), 1249
WGate (class in sympy.physics.quantum.grover), 1555
where() (in module sympy.stats), 1253
whitm() (in module mpmath), 768
whitw() (in module mpmath), 769
wicks() (in module sympy.physics.secondquant), 1411
width (sympy.categories.diagram_drawing.DiagramGrid attribute), 1574
width() (sympy.printing.pretty.stringpict.StringPict method), 1130
Wigner3j (class in sympy.physics.quantum.cg), 1512
Wigner6j (class in sympy.physics.quantum.cg), 1512
Wigner9j (class in sympy.physics.quantum.cg), 1512
wigner_3j() (in module sympy.physics.wigner), 1421
wigner_6j() (in module sympy.physics.wigner), 1422
wigner_9j() (in module sympy.physics.wigner), 1423
WignerD (class in sympy.physics.quantum.spin), 1534
WignerSemicircle() (in module sympy.stats), 1250
Wild (class in sympy.core.symbol), 117
WildFunction (class in sympy.core.function), 149
workdps() (in module mpmath), 650
workprec() (in module mpmath), 650
write() (sympy.utilities.codegen.CodeGen method), 1342
write() (sympy.utilities.runtests.PyTestReporter method), 1371
wronskian() (in module sympy.matrices.dense), 591

X

X (in module sympy.physics.quantum.gate), 1553
x (sympy.geometry.point.Point attribute), 446
x (sympy.physics.mechanics.essential.ReferenceFrame attribute), 1479
XBra (class in sympy.physics.quantum.cartesian), 1518
xdirection (sympy.geometry.line.Ray attribute), 457
XGate (class in sympy.physics.quantum.gate), 1552
XKet (class in sympy.physics.quantum.cartesian), 1518
xobj() (in module sympy.printing.pretty.pretty_symbology), 1128
XOp (class in sympy.physics.quantum.cartesian), 1518
Xor (class in sympy.logic.boolalg), 544
xreplace() (sympy.core.basic.Basic method), 96
xrand() (in module sympy.printing.pretty.pretty_symbology), 1128
xsym() (in module sympy.printing.pretty.pretty_symbology), 1129
xthreaded() (in module sympy.utilities.decorator), 1346
xypic_draw_diagram() (in module sympy.categories.diagram_drawing), 1578
XypicDiagramDrawer (class in sympy.categories.diagram_drawing), 1576

Y

Y (in module sympy.physics.quantum.gate), 1553
y (sympy.geometry.point.Point attribute), 446
y (sympy.physics.mechanics.essential.ReferenceFrame attribute), 1479
ydirection (sympy.geometry.line.Ray attribute), 458
YGate (class in sympy.physics.quantum.gate), 1552
yn (class in sympy.functions.special.bessel), 366
Ynm (class in sympy.functions.special.spherical_harmonics), 391
Ynm_c() (in module sympy.functions.special.spherical_harmonics), 393
YOp (class in sympy.physics.quantum.cartesian), 1518

Z

Z (in module sympy.physics.quantum.gate), 1553
z (sympy.physics.mechanics.essential.ReferenceFrame attribute), 1479

zero (sympy.polys.polytools.Poly attribute), 999
ZeroMatrix (class in sympy.matrices.expressions), 618
zeros() (in module sympy.matrices.dense), 588
zeros() (sympy.matrices.sparse.SparseMatrix class method), 607
zeta (class in sympy.functions.special.zeta_functions), 368
zeta() (in module mpmath), 836
zetazero() (in module mpmath), 843
ZGate (class in sympy.physics.quantum.gate), 1552
zip_row_op() (sympy.matrices.dense.MutableDenseMatrix method), 599
zip_row_op() (sympy.matrices.sparse.MutableSparseMatrix method), 611
Znm (class in sympy.functions.special.spherical_harmonics), 393
zoo() (in module sympy.core.numbers), 126
ZOp (class in sympy.physics.quantum.cartesian), 1518