



Diofant Documentation

Release 0.15.0a1.dev106+g2def774

Diofant Development Team

May 12, 2024

CONTENTS

1	Installation	1
1.1	From Sources	1
1.2	Run Diofant	1
1.3	Feedback	2
2	Tutorial	3
2.1	Introduction	3
2.2	Basics	6
2.3	Gotchas	9
2.4	Printing	10
2.5	Simplification	12
2.6	Calculus	18
2.7	Solvers	23
2.8	Polynomials	25
2.9	Matrices	28
2.10	Expression Trees	34
3	Command-Line Usage	39
3.1	python -m diofant	39
4	Reference	41
4.1	Config	41
4.2	Core	41
4.3	Combinatorics	142
4.4	Number Theory	228
4.5	Concrete Mathematics	261
4.6	Mathematical Functions	274
4.7	Integrals	392
4.8	Logic	417
4.9	Domains	426
4.10	Matrices	432
4.11	Polynomials	505
4.12	Printing	548
4.13	Interactive	566
4.14	Sets	567
4.15	Simplify	580
4.16	Solvers	606
4.17	Tensors	696
4.18	Utilities	710
4.19	Parsing	748

4.20 Calculus	752
5 Internals	757
5.1 Internals of the Polynomial Manipulation Module	757
5.2 The Gruntz Algorithm	787
5.3 Details on the Hypergeometric Function Expansion	789
5.4 Computing Integrals using Meijer G-Functions	798
5.5 Numerical evaluation	812
6 Development	819
6.1 Reporting Issues	819
6.2 Contributing Code	819
6.3 Rosetta Stone	820
6.4 Versioning	821
6.5 Release Procedure	821
7 About	823
7.1 License	823
8 Release Notes	825
8.1 Diofant 0.15	825
8.2 Diofant 0.14	828
8.3 Diofant 0.13	832
8.4 Diofant 0.12	836
8.5 Diofant 0.11	839
8.6 Diofant 0.10	844
8.7 Diofant 0.9	849
8.8 Diofant 0.8	857
8.9 SymPy releases	865
Bibliography	899
Python Module Index	903
Index	905

INSTALLATION

The Diofant can be installed on any computer with Python 3.11 or above. You can install latest release with pip:

```
pip install diofant
```

or to install also extra dependencies:

```
pip install diofant[gmpy,interactive]
```

To use *Unicode pretty printing* (page 11) — configure your system to have good TTF fonts. The *DejaVu Sans Mono* seems to be an acceptable choice. On Debian you can install this [font package](#) with:

```
apt install fonts-dejavu
```

1.1 From Sources

If you are a developer or like to get the latest updates as they come, be sure to install from the git repository and include required extra dependencies:

```
git clone git://github.com/diofant/diofant.git
cd diofant
pip install -e .[develop,docs,tests]
```

1.2 Run Diofant

To verify that your freshly-installed Diofant works, please start up the Python interpreter:

```
python
```

and execute some simple statements like the ones below:

```
>>> from diofant.abc import x
>>> ((1 + x)**(1/x)).limit(x, 0)
E
```

Tip: *Run Diofant as a module* (page 39) for interactive work:

```
python -m diofant
```

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

Also, you may want to run full set of unit tests to make sure everything works:

```
pytest --pyargs diofant
```

`pytest` and some other packages are required for testing, so be sure to install the Diofant first with the optional “tests” list of dependencies:

```
pip install diofant[tests]
```

1.3 Feedback

If you think there’s a bug, you have a question or you would like to request a feature, please [open an issue ticket](#) (page 819). General questions and comments can be [sent](#) to the [Diofant mailing list](#).

TUTORIAL

Warning: It is assumed that the reader already knows the Python programming language. If you do not, please start from the [Python tutorial](#).

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

2.1 Introduction

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. Start the Python interpreter:

```
python
```

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
```

Here we got the exact answer — 9 is a perfect square — but usually it will be an approximate result

```
>>> math.sqrt(8)
2.8284271247461903
```

This is where symbolic computation first comes in: with a symbolic computation system like Diofant, square roots of numbers that are not perfect squares are left unevaluated by default

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

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

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

Yet we can also approximate this number with any precision

```
>>> .evalf(20)
2.8284271247461900976
```

The above example starts to show how we can manipulate irrational numbers exactly using Diofant. Now we introduce symbols.

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

```
>>> x, y = diofant.symbols('x y')
>>> expr = x + 2*y
>>> expr
x + 2*y
```

Note: Unlike many symbolic manipulation systems you may have used, in Diofant symbols are not defined automatically. To define symbols (instances of [Symbol](#) (page 80)) you may use [symbols\(\)](#) (page 82).

Note that we wrote $x + 2y$, using Python’s mathematical syntax, 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 $x + 2y$. 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 $x + 2y - x$, but rather just $2y$. The x and the $-x$ automatically canceled one another. This is similar to how `sqrt(8)` automatically turned into $2\sqrt{2}$ above.

Tip: Use [evaluate\(\)](#) (page 56) context or `evaluate` flag to prevent automatic evaluation, for example:

```
>>> diofant.sqrt(8, evaluate=False)
sqrt(8)
>>> _.doit()
2*sqrt(2)
```

This isn’t always the case in Diofant, 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 Diofant. 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 Diofant, there are functions to go from one form to the other

```
>>> diofant.expand(x*expr)
x**2 + 2*x*y
```

(continues on next page)

(continued from previous page)

```
>>> diofant.factor(_)
x*(x + 2*y)
```

The real power of a symbolic computation system (which by the way, are also often called computer algebra systems, or just CASs) such as Diofant is the ability to do all sorts of computations symbolically: simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, and much more. Diofant includes modules for printing (like 2D pretty printed output of math formulas, or \LaTeX), code generation, combinatorics, number theory, logic, and more. Here is a small sampling of the sort of symbolic power Diofant is capable of, to whet your appetite.

Note: From here on in this tutorial we assume that these statements were executed:

```
>>> from diofant import *
>>> a, b, c, d, t, x, y, z = symbols('a:d t x:z')
>>> init_printing(pretty_print=True)
```

Last one will make all further examples pretty print with unicode characters.

`import *` has been used here to aid the readability of the tutorial, but is best to avoid such wildcard import statements in production code, as they make it unclear which names are present in the namespace.

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

```
>>> diff(sin(x)*exp(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
e · sin(x)
```

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

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

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

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

Solve $x^2 - 2 = 0$.

```
>>> solve(x**2 - 2, x)
[[{x: -√ 2}, {x: √ 2}]]
```

Solve the differential equation $f'' - f = e^x$.

```
>>> f = symbols('f', cls=Function)
>>> dsolve(Eq(f(x).diff((x, 2)) - f(x), exp(x)))
x      x      -x
f(x) = e · (C2 + -) + e · C1
2
```

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

```
>>> Matrix([[1, 2], [2, 2]]).eigenvals()
{ - 3 +  $\frac{\sqrt{17}}{2}$ : 1, - 3 -  $\frac{\sqrt{17}}{2}$ : 1 }
```

Rewrite the Bessel function $J_y(z)$ in terms of the spherical Bessel function $j_y(z)$.

```
>>> besselj(y, z).rewrite(jn)
 $\frac{\sqrt{2} \cdot \sqrt{z} \cdot j_n(y - 1/2, z)}{\sqrt{\pi}}$ 
```

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

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

2.2 Basics

Here we discuss some of the most basic aspects of expression manipulation in Diofant.

2.2.1 Assumptions

The assumptions system allows users to declare certain mathematical properties on symbols, such as being positive, imaginary or integer.

By default, all symbols are complex valued. This assumption makes it easier to treat mathematical problems in full generality.

```
>>> sqrt(x**2)
 $\sqrt{x^2}$ 
```

Yet obviously we can simplify above expression if some additional mathematical properties on x are assumed. This is where assumptions system come into play.

Assumptions are set on [Symbol](#) (page 80) objects when they are created. For instance, we can create a symbol that is assumed to be positive.

```
>>> p = symbols('p', positive=True)
```

And then, certain simplifications will be possible:

```
>>> sqrt(p**2)
p
```

The assumptions system additionally has deductive capabilities. You might check assumptions on any expression with `is_assumption` attributes, like `is_positive` (page 75).

```
>>> p.is_positive
True
>>> (1 + p).is_positive
True
>>> (-p).is_positive
False
```

Note: False is returned also if certain assumption doesn't make sense for given object.

In a three-valued logic, used by system, None represents the “unknown” case.

```
>>> (p - 1).is_positive is None
True
```

2.2.2 Substitution

One of the most common things you might want to do with a mathematical expression is substitution with `subs()` (page 52) method. It replaces all instances of something in an expression with something else.

```
>>> expr = cos(x) + 1
>>> expr.subs({x: y})
cos(y) + 1
>>> expr
cos(x) + 1
```

We see that performing substitution leaves original expression `expr` unchanged.

Note: Almost all Diofant expressions are immutable. No function (or method) will change them in-place.

Use several method calls to perform a sequence of substitutions in same variable:

```
>>> x**y
xy
>>> .subs({y: x**y}).subs({y: x**x})
x(xx)
```

Use flag `simultaneous` to do all substitutions at once.

```
>>> (x - y).subs({x: y, y: x})
0
>>> (x - y).subs({x: y, y: x}, simultaneous=True)
-x + y
```

2.2.3 Numerics

To evaluate a numerical expression into a floating point number with arbitrary precision, use `evalf()` (page 138). By default, 15 digits of precision are used.

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

But you can change that. Let's compute the first 70 digits of π .

```
>>> pi.evalf(70)
3.141592653589793238462643383279502884197169399375105820974944592307816
```

Complex numbers are supported:

```
>>> (1/(pi + I)).evalf()
0.289025482222236 - 0.0919996683503752*I
```

If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `evalf()` (page 138) returns the original expression or a partially evaluated expression.

```
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x2 + 0.333333333333333*x
```

You can also use the standard Python functions `float` and `complex` to convert symbolic expressions to regular Python numbers:

```
>>> float(pi)
3.141592653589793
>>> complex(pi + E*I)
(3.141592653589793+2.718281828459045j)
```

Sometimes there are roundoff errors smaller than the desired precision that remain after an expression is evaluated. Such numbers can be removed by setting the chop flag.

```
>>> one = cos(1)**2 + sin(1)**2
>>> (one - 1).evalf(strict=False)
-0.e-146
>>> (one - 1).evalf(chop=True)
0
```

Discussed above method is not effective enough if you intend to evaluate an expression at many points, there are better ways, especially if you only care about machine precision.

Substitution may be used to evaluate an expression for some floating point number

```
>>> expr = sin(x)/x
>>> expr.subs({x: 0.1})
0.998334166468282
```

but this method is slow.

The easiest way to convert an expression to the form that can be numerically evaluated with libraries like `numpy` or the standard library `math` module — use the `lambdify()` (page 745) function.

```
>>> f = lambdify(x, expr, 'math')
>>> f(0.1)
0.9983341664682815
```

Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```
>>> f = lambdify(x, expr, 'numpy')
>>> f(range(1, 5))
[ 0.84147098  0.45464871  0.04704    -0.18920062]
```

2.3 Gotchas

Lets recall again, that Diofant is nothing more than a Python library, like `numpy` or even the Python standard library module `sys`. What this means is that Diofant does not add anything to the Python language. Limitations that are inherent in the language are also inherent in Diofant.

In this section we are trying to collect some things that could surprise newcomers.

2.3.1 Numbers

To begin with, it should be clear for you, that if you type a numeric literal — it will create a Python number of type `int` or `float`.

Diofant uses its own classes for numbers, for example `Integer` (page 89) instead of `int`. In most cases, Python numeric types will be correctly coerced to Diofant numbers during expression construction.

```
>>> 3 + x**2

$$x^2 + 3$$

>>> type(_ - x**2)
<class 'diofant.core.numbers.Integer'>
```

But if you use some arithmetic operators between two numerical literals, Python will evaluate such expression before Diofant has a chance to get to them.

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

$$x^{1.5}$$

```

Tip: Wrapping the integer division with `Fraction` is automatically enabled if you *run Diofant as a module* (page 39).

The universal solution is using correct Diofant numeric class to construct numbers explicitly. For example, `Rational` (page 88) in the above example

```
>>> x**Rational(3, 2)

$$x^{3/2}$$

```

2.3.2 Equality

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

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

But, $(x + 1)^2$ *does* equal $x^2 + 2x + 1$. What is going on here?

In Diofant, `==` represents structural equality testing and $(x + 1)^2$ and $x^2 + 2x + 1$ are not the same in this sense. One is the power and the other is the addition of three terms.

There is a separate class, called [Eq](#) (page 107), which can be used to create a symbolic equation

```
>>> Eq((x + 1)**2 - x**2, 2*x + 1)
- x2 + (x + 1)2 = 2·x + 1
```

It is not always return a `bool` object, like `==`, but you may use some simplification methods to prove (or disprove) equation.

```
>>> expand(_)
true
```

2.3.3 Naming of Functions

Diofant uses different names for some mathematical functions than most computer algebra systems. In particular, the inverse trigonometric functions use the python names of [asin\(\)](#) (page 285), [acos\(\)](#) (page 286) and so on instead of `arcsin` and `arccos`.

2.4 Printing

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

- [Str](#) (page 11)
- [Repr](#) (page 11)
- [2D Pretty Printer](#) (page 11)
- [LaTeX](#) (page 12)
- [Dot](#) (page 12)

In addition to these, there are also “printers” that can output Diofant objects to code, such as C, Fortran, or Mathematica.

Best printer is enabled automatically for interactive session (i.e. \LaTeX in the IPython notebooks, pretty printer in the IPython console or str printer in the Python console). If you want manually configure pretty printing, please use the [init_printing\(\)](#) (page 566) function.

Lets take this simple expression

2.4.1 Str

```
>>> str(expr)
'Integral(sqrt(1/x), x)'
>>> print(expr)
Integral(sqrt(1/x), x)
```

```
>>> repr(expr)
"Integral(Pow(Pow(Symbol('x'), Integer(-1)), Rational(1, 2)), Tuple(Symbol('x')))"
```

```
>>> pretty(expr)
'\int \sqrt{x} dx'
>>> print(_)
```

$$\int \sqrt{x} \, dx$$

```
>>> pprint(expr)
```

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

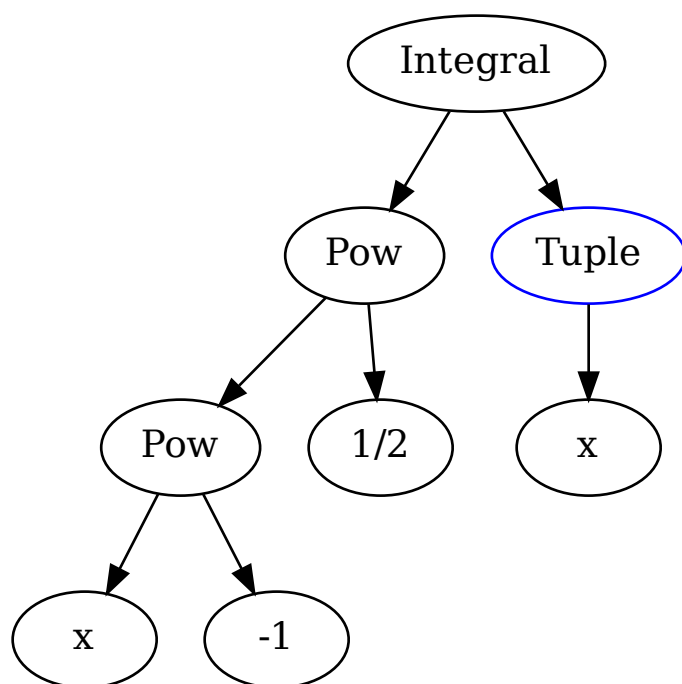
2.4.4 LaTeX

To get the \LaTeX form of an expression, use `latex()` (page 558).

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

2.4.5 Dot

`dotprint()` (page 565) function prints output to dot format, which can be rendered with Graphviz:



2.5 Simplification

The generic way to do *nontrivial* simplifications of expressions is calling `simplify()` (page 580) function.

```
>>> 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)
```


There are also more directed simplification functions. These apply very specific rules to the input expression and are typically able to make guarantees about the output. For instance, the `factor()` (page 530) function, given a polynomial with rational coefficients in several variables, is guaranteed to produce a factorization into irreducible factors.

The `simplify()` (page 580) function applies almost all available in Diofant such specific simplification rules in some heuristics sequence to produce the simplest result.

Tip: The optional `measure` keyword argument for `simplify()` (page 580) lets the user specify the Python function used to determine how “simple” an expression is. The default is `count_ops()` (page 133), which returns the total number of operations in the expression.

That is why it is usually slow. But more important pitfall is that sometimes `simplify()` (page 580) doesn’t “simplify” how you might expect, if, for example, it miss some transformation or apply it too early or too late. Lets look on an example

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

$$x^2 + 2 \cdot x + 1$$

>>> factor(_)

$$(x + 1)^2$$

```

Obviously, the factored form is more “simple”, as it has less arithmetic operations.

The function `simplify()` (page 580) 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()` (page 580) 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.

2.5.1 Rational Functions

`expand()` (page 129) is one of the most common simplification functions in Diofant. Although it has a lot of scopes, for now, we will consider its function in expanding polynomial expressions.

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

$$x^2 + 2 \cdot x + 1$$

>>> expand((x + 2)*(x - 3))

$$x^2 - x - 6$$

```

Given a polynomial, `expand()` (page 129) will put it into a canonical form of a sum of monomials with help of more directed expansion methods, namely `expand_multinomial()` (page 136) and `expand_mul()` (page 134).

`expand()` (page 129) 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()` (page 129) on it due to cancellation.

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

Function `factor()` (page 530) takes a multivariate polynomial with rational coefficients and factors it into irreducible factors.

```
>>> factor(x**3 - x**2 + x - 1)
(x - 1)·(x2 + 1)
>>> factor(x**2*z + 4*x*y*z + 4*y**2*z)
z·(x + 2·y)2
```

For polynomials, `factor()` (page 530) is the opposite of `expand()` (page 129).

Note: The input to `factor()` (page 530) and `expand()` (page 129) need not be polynomials in the strict sense. They will intelligently factor or expand any kind of expression (though, for example, the factors may not be irreducible if the input is no longer a polynomial over the rationals).

```
>>> expand((cos(x) + sin(x))**2)
sin2(x) + 2·sin(x)·cos(x) + cos2(x)
>>> factor(_)
(sin(x) + cos(x))2
```

`collect()` (page 588) collects common powers of a term in an expression.

```
>>> x*y + x - 3 + 2*x**2 - z*x**2 + x**3
x3 - x2·z + 2·x2 + x·y + x - 3
>>> collect(_, x)
x3 + x2·(-z + 2) + x·(y + 1) - 3
```

`collect()` (page 588) is particularly useful in conjunction with the `coeff()` (page 65) method.

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

`cancel()` (page 527) will take any rational function and put it into the standard canonical form, p/q , where p and q are expanded polynomials with no common factors.

```
>>> 1/x + (3*x/2 - 2)/(x - 4)
3·x
— - 2
2
x - 4
+ —
x
>>> cancel(_)
3·x2 - 2·x - 8
—
2·x2 - 8·x
```

```
>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
x·y2 - 2·x·y·z + x·z2 + y2 - 2·y·z + z2
—
x2 - 1
>>> cancel(_)
y2 - 2·y·z + z2
—
x - 1
```

Note: Since `factor()` (page 530) will completely factorize both the numerator and the denominator of an expression, it can also be used to do the same thing:

```
>>> factor(expr)

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

```

However, it's less efficient if you are only interested in making sure that the expression is in canceled form.

`apart()` (page 545) performs a [partial fraction decomposition](#) on a rational function.

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

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

>>> apart(_)

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

```

2.5.2 Trigonometric Functions

To simplify expressions using trigonometric identities, use `trigsimp()` (page 593) function.

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

$$\frac{\cos(4 \cdot x)}{2} + \frac{1}{2}$$

>>> trigsimp(sin(x)*tan(x)/sec(x))

$$\sin^2(x)$$

```

It also works with hyperbolic functions.

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

Much like `simplify()` (page 580) function, `trigsimp()` (page 593) applies various trigonometric identities to the input expression, and then uses a heuristic to return the “best” one.

To expand trigonometric functions, that is, apply the sum or double angle identities, use `expand_trig()` (page 135) function.

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

$$\frac{2 \cdot \tan(x)}{1 - \tan^2(x) + 1}$$

```

2.5.3 Powers and Logarithms

`powdenest()` (page 597) function applies identity $(x^a)^b = x^{ab}$, from left to right, if assumptions allow.

```
>>> a, b = symbols('a b', real=True)
>>> p = symbols('p', positive=True)
>>> powdenest((p**a)**b)
pa·b
```

`powsimp()` (page 596) function reduces expression by combining powers with similar bases and exponent.

```
>>> powsimp(z**x*z**y)
zx + y
```

Again, as for `powdenest()` (page 597) above, for the identity $x^a y^a = (xy)^a$, that combine bases, we should be careful about assumptions.

```
>>> q = symbols('q', positive=True)
>>> powsimp(p**a*q**a)
(p·q)a
```

In general, this identity doesn't hold. For example, if $x = y = -1$ and $a = 1/2$.

`expand_power_exp()` (page 136) and `expand_power_base()` (page 137) functions do reverse of `powsimp()` (page 596).

```
>>> expand_power_exp(x**(y + z))
xy·xz
>>> expand_power_base((p*q)**a)
pa·qa
```

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.

To apply above identities from left to right, use `expand_log()` (page 135). As for powers, the identities will not be applied unless they are valid with given set of assumptions for symbols.

```
>>> expand_log(log(p*q))
log(p) + log(q)
>>> expand_log(log(p/q))
log(p) - log(q)
>>> expand_log(log(p**2))
2·log(p)
>>> expand_log(log(p**a))
a·log(p)
>>> expand_log(log(x*y))
log(x·y)
```

To apply identities from right to left, i.e. do reverse of `expand_log()` (page 135), use `logcombine()` (page 586) function.

```
>>> logcombine(log(p) + log(q))
log(p·q)
>>> logcombine(a*log(p))
```

(continues on next page)

(continued from previous page)

```

log( $\frac{a}{p}$ )
>>> logcombine(a*log(z))
a*log(z)

```

2.5.4 Special Functions

Diofant implements dozens of *special functions* (page 318), ranging from functions in combinatorics to mathematical physics.

To expand special functions in terms of some identities, use `expand_func()` (page 135). For example the *gamma function* `gamma` (page 319) can be expanded as

```

>>> expand_func(gamma(x + 3))
x*(x + 1)*Gamma(x + 2)

```

This method also can help if you would like to rewrite the generalized hypergeometric function *hyper* (page 367) or the Meijer G-function *meijerg* (page 369) in terms of more standard functions.

```

>>> expand_func(hyper([1, 1], [2], z))
-log(-z + 1)
>>> meijerg([1], [1], [1], [], -z)
G1, 11, 1(1, 1 | -z)
>>> expand_func(_)
z
√e

```

Another type of expand rule is expanding complex valued expressions and putting them into a normal form. For this `expand_complex()` (page 136) is used. Note that it will always perform arithmetic expand to obtain the desired normal form.

```

>>> expand_complex(x + I*y)
i*(re(y) + im(x)) + re(x) - im(y)

```

The same behavior can be obtained by using `as_real_imag()` (page 64) method.

```

>>> (x + I*y).as_real_imag()
(re(x) - im(y), re(y) + im(x))

```

To simplify combinatorial expressions, involving *factorial* (page 310), *binomial* (page 306) or *gamma* (page 319) — use `combsimp()` (page 599) function.

```

>>> combsimp(factorial(x)/factorial(x - 3))
x*(x - 2)*(x - 1)
>>> combsimp(binomial(x + 1, y + 1)/binomial(x, y))
x + 1
y + 1
>>> combsimp(gamma(x)*gamma(1 - x))
π
sin(π*x)

```

2.5.5 CSE

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is called common subexpression elimination (CSE) and implemented in the `cse()` (page 601) function.

```
>>> cse(sqrt(sin(x)))
([], [sqrt(sin(x))])
```

```
>>> cse(sqrt(sin(x) + 5)*sqrt(sin(x) + 4))
([(x0, sin(x))], [sqrt(x0 + 4) * sqrt(x0 + 5)])
```

```
>>> cse(sqrt(sin(x + 1) + 5 + cos(y))*sqrt(sin(x + 1) + 4 + cos(y)))
([(x0, sin(x + 1) + cos(y))], [sqrt(x0 + 4) * sqrt(x0 + 5)])
```

```
>>> cse((x - y)*(z - y) + sqrt((x - y)*(z - y)))
([(x0, -y), (x1, (x + x0)*(x0 + z))], [sqrt(x1 + x1)])
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the `optimizations` optional argument.

```
>>> cse((x - y)*(z - y) + sqrt((x - y)*(z - y)), optimizations='basic')
([(x0, -(x - y)*(y - z))], [sqrt(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.

2.6 Calculus

This section covers how to do basic calculus tasks such as derivatives, integrals, limits, and series expansions in Diofant.

2.6.1 Derivatives

To take derivatives, use the `diff()` (page 125) function.

```
>>> diff(cos(x))
-sin(x)
>>> diff(exp(x**2), x)
(2)
(x)
2 * ex2 * x
```

`diff()` (page 125) can take multiple derivatives at once. To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a tuple (variable and order). 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)
x·y·z 3 2 ( 3 3 3 2 2 2
e      ·x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)
>>> diff(expr, x, (y, 2), (z, 4))
x·y·z 3 2 ( 3 3 3 2 2 2
e      ·x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)
>>> diff(expr, x, y, y, (z, 4))
x·y·z 3 2 ( 3 3 3 2 2 2
e      ·x·y·(x·y·z + 14·x·y·z + 52·x·y·z + 48)
```

`diff()` (page 125) can also be called as a method `diff()` (page 67). The two ways of calling `diff()` (page 125) are exactly the same, and are provided only for convenience.

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

To create an unevaluated derivative, use the `Derivative` (page 122) class. It has the same syntax as `diff()` (page 125).

```
>>> Derivative(expr, x, y, y, (z, 4))
7
∂ (x·y·z)
-----
∂z4 ∂y2 ∂x e
```

Such unevaluated objects also used when Diofant does not know how to compute the derivative of an expression.

To evaluate an unevaluated derivative, use the `doit()` (page 47) method.

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

2.6.2 Integrals

To compute an integral, use the `integrate()` (page 399) function. There are two kinds of integrals, definite and indefinite. To compute an indefinite integral, do

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

Note: For indefinite integrals, Diofant does not include the constant of integration.

For example, to compute a definite integral

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

we would do

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

Tip: ∞ in Diofant is `oo` (that's the lowercase letter “oh” twice).

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()` (page 399) is unable to compute an integral, it returns an unevaluated *Integral* (page 405) object.

```
>>> integrate(x**x)

$$\int x^x dx$$

>>> print(_)
Integral(x**x, x)
```

As with *Derivative* (page 122), you can create an unevaluated integral directly. To later evaluate this integral, call `doit()` (page 406).

```
>>> Integral(log(x)**2)

$$\int \log^2(x) dx$$

>>> _.doit()
x·log(x)2 - 2·x·log(x) + 2·x
```

`integrate()` (page 399) uses powerful algorithms that are always improving to compute both definite and indefinite integrals, including a partial implementation of the *Risch algorithm*

```
>>> 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)))

$$\int \frac{x \left( x^2 e^x - 2 e^x x - e^x + x^4 - x^2 - 2x \right)}{\left( e^x + 1 \right) \cdot (x - 1)^2 \cdot (x + 1)^2} dx$$

>>> x.doit()
```

(continues on next page)

(continued from previous page)

$$\frac{e}{x^2 - 1} + \log\left(\frac{x}{e^x + 1}\right)$$

and an algorithm using [Meijer G-functions](#) that is useful for computing integrals in terms of special functions, especially definite integrals

```
>>> Integral(sin(x**2))

$$\int \sin\left(x^2\right) dx$$

>>> _.doit()

$$\frac{3 \cdot \sqrt{2} \cdot \sqrt{\pi} \cdot \operatorname{fresnels}\left(\frac{\sqrt{2} \cdot x}{\sqrt{\pi}}\right) \cdot \Gamma(3/4)}{8 \cdot \Gamma(7/4)}$$

```

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

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

>>> _.doit()

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

```

This last example returned a [Piecewise](#) (page 297) expression because the integral does not converge unless $\Re(y) > -1$.

2.6.3 Sums and Products

Much like integrals, there are [summation\(\)](#) (page 271) and [product\(\)](#) (page 272) to compute sums and products respectively.

```
>>> summation(2**x, (x, 0, y - 1))

$$2^y - 1$$

>>> product(z, (x, 1, y))

$$z^y$$

```

Unevaluated sums and products are represented by [Sum](#) (page 262) and [Product](#) (page 265) classes.

```
>>> Sum(1, (x, 1, z))

$$\sum_{x=1}^z 1$$

```

(continues on next page)

(continued from previous page)

```

\
1
x = 1
>>> _ .doit()
z

```

2.6.4 Limits

Diofant can compute symbolic limits with the `limit()` (page 753) function. To compute a directional limit

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

do

```

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

```

`limit()` (page 753) should be used instead of `subs()` (page 52) whenever the point of evaluation is a singularity. Even though Diofant 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* (page 122) and *Integral* (page 405), `limit()` (page 753) has an unevaluated counterpart, `Limit` (page 752). To evaluate it, use `doit()` (page 753).

```

>>> Limit((cos(x) - 1)/x, x, 0)
cos(x) - 1
lim
x -> 0+
>>> _ .doit()
0

```

To change side, pass '-' as a third argument to `limit()` (page 753). For example, to compute

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

do

```

>>> limit(1/x, x, 0, dir=1)
-oo

```

You can also evaluate bidirectional limit

```

>>> limit(sin(x)/x, x, 0, dir=Reals)
1
>>> limit(1/x, x, 0, dir=Reals)
Traceback (most recent call last):
PoleError: left and right limits for expression 1/x at point x=0 seems to be not equal

```

2.6.5 Series Expansion

Diofant can compute asymptotic series expansions of functions around a point.

```
>>> exp(sin(x)).series(x, 0, 4)

$$1 + x + \frac{x^2}{2} + O(x^4)$$

```

The $O(x^4)$ term, an instance of `O` (page 753) 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 $x = \infty$). Order terms can be created and manipulated outside of `series()` (page 79).

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

$$x + x^3 + x^6 + O(x^4)$$

>>> x*O(1, x)

$$O(x)$$

```

If you do not want the order term, use the `removeO()` (page 78) method.

```
>>> exp(x).series(x, 0, 3).removeO()

$$\frac{x^2}{2} + x + 1$$

```

The `O` (page 753) notation supports arbitrary limit points:

```
>>> exp(x - 1).series(x, x0=1)

$$\frac{(x-1)^2}{2} + \frac{(x-1)^3}{6} + \frac{(x-1)^4}{24} + \frac{(x-1)^5}{120} + x + O((x-1)^6; x \rightarrow 1)$$

```

2.7 Solvers

This section covers equations solving.

Note: Any expression in input, that not in an `Eq` (page 107) is automatically assumed to be equal to 0 by the solving functions.

2.7.1 Algebraic Equations

The main function for solving algebraic equations is `solve()` (page 606).

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

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

If no solutions are found, an empty list is returned.

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

`solve()` (page 606) can also solve systems of equations.

```
>>> solve([x - y + 2, x + y - 3])
[{x: 1/2, y: 5/2}]
>>> solve([x*y - 7, x + y - 6])
[[{x: -sqrt(2) + 3, y: sqrt(2) + 3}, {x: sqrt(2) + 3, y: -sqrt(2) + 3}]
```

Each solution reported only once:

```
>>> solve(x**3 - 6*x**2 + 9*x)
[{x: 0}, {x: 3}]
```

To get the solutions of a polynomial including multiplicity use `roots()` (page 542).

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

2.7.2 Recurrence Equations

To solve recurrence equations, use `rsolve()` (page 684). First, create an undefined function by passing `cls=Function` to the `symbols()` (page 82) function.

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

We can call `f(x)`, and it will represent an unknown function application.

Note: From here on in this tutorial we assume that these statements were executed:

```
>>> from diofant import *
>>> a, b, c, d, t, x, y, z = symbols('a:d t x:z')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f:h', cls=Function)
>>> init_printing(pretty_print=True)
```

As for algebraic equations, the output is a list of `dict`'s

```
>>> rsolve(f(n + 1) - 3*f(n) - 1)
[[{f: n -> 3**n * C0 - 1/2}]]
```

The arbitrary constants in the solutions are symbols of the form `C0`, `C1`, and so on.

2.7.3 Differential Equations

To solve the differential equation

```
>>> Eq(f(x).diff(x, x) - 2*f(x).diff(x) + f(x), sin(x))
f(x) - 2 * d/dx(f(x)) + d^2/dx^2(f(x)) = sin(x)
```

Note: Derivatives of the unknown function $f(x)$ are unevaluated.

we would use

```
>>> dsolve(_)
f(x) = ex · (C1 + C2 · x) +  $\frac{\cos(x)}{2}$ 
```

`dsolve()` (page 635) can also solve systems of equations, like `solve()` (page 606).

```
>>> dsolve([f(x).diff(x) - g(x), g(x).diff(x) - f(x)])
[ $f(x) = e^x \cdot C_2 - e^{-x} \cdot C_1$ ,  $g(x) = e^x \cdot C_2 + e^{-x} \cdot C_1$ ]
```

2.8 Polynomials

We show here some functions, that provide different algorithms dealing with polynomials in the form of Diofant expression.

Note, that coefficients of a polynomial can be elements of any commutative ring, this ring is called the domain of the polynomial ring and may be specified as a keyword parameter for functions. Polynomial generators also can be specified via an arbitrary number of arguments after required arguments of functions.

2.8.1 Division

The function `div()` (page 529) provides division of polynomials with remainder. That is, for polynomials f and g , it computes q and r , such that $f = g \cdot q + r$ and $\deg(r) < \deg(g)$. For polynomials in one variables with coefficients in a field, say, the rational numbers, q and r are uniquely defined this way

```
>>> f, g = 5*x**2 + 10*x + 3, 2*x + 2
```

```
>>> div(f, g)
( $\frac{5 \cdot x}{2} + \frac{5}{2}$ , -2)
>>> expand(_[0]*g + _[1])
5 · x2 + 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

```
>>> div(f, g, field=False)
(5, 5 · x2 - 7)
```

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
```

```
>>> div(f, g, field=False)
(x, 9*x + 3)
>>> expand(_[0]*g + _[1])
5*x2 + 10*x + 3
```

This also works for polynomials with multiple variables

```
>>> div(x*y + y*z, 3*x + 3*z)
(y
 -, 0)
 3)
```

2.8.2 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

```
>>> gcd((12*x + 12)*x, 16*x**2)
4*x
```

But if the polynomials have rational coefficients, then the returned polynomial is monic

```
>>> gcd(3*x**2/2, 9*x/4)
x
```

Symbolic exponents are supported

```
>>> gcd(2*x**(n + 4) - x**(n + 2), 4*x**(n + 1) + 3*x**n)
xn
```

It also works with multiple variables. In this case, the variables are ordered alphabetically, by default, which has influence on the leading coefficient

```
>>> gcd(x*y/2 + y**2, 3*x + 6*y)
x + 2*y
```

The lcm is connected with the gcd and one can be computed using the other

```
>>> f, g = x*y**2 + x**2*y, x**2*y**2
```

```
>>> gcd(f, g)
x*y
>>> lcm(f, g)
x3.y2 + x2.y3
>>> expand(f*g)
x4.y3 + x3.y4
>>> expand(gcd(f, g, x, y)*lcm(f, g, x, y))
x4.y3 + x3.y4
```

2.8.3 Square-free factorization

The square-free factorization of a polynomial is a factorization into powers of square-free polynomials (i.e. not divisible by any square of a non-constant polynomial)

```
>>> sqf(2*x**2 + 5*x**3 + 4*x**4 + x**5)
(x + 2) * (x**2 + x) ** 2
>>> sqf(x**5*y**5 + 1, modulus=5)
(x*y + 1) ** 5
```

2.8.4 Factorization

Factorization supported over different domains, lets compute one for the rational field, its algebraic extension or the finite field of order 5

```
>>> f = x**4 - 3*x**2 + 1
```

```
>>> factor(f)
(x**2 - x - 1) * (x**2 + x - 1)
>>> factor(f, extension=GoldenRatio)
(x - phi) * (x + phi) * (x - 1 + phi) * (x - phi + 1)
>>> factor(f, modulus=5)
(x + 2) ** 2 * (x + 3) ** 2
```

The finite fields of prime power order are supported

```
>>> factor(f, modulus=8)
(x + 4) ** 2 * (x + 7) ** 2
```

You also may use gaussian keyword to obtain a factorization over Gaussian rationals

```
>>> factor(4*x**4 + 8*x**3 + 77*x**2 + 18*x + 153, gaussian=True)
4 * (x - 3*i/2) * (x + 3*i/2) * (x + 1 - 4*i) * (x + 1 + 4*i)
```

Computing with multivariate polynomials over various domains is as simple as in univariate case.

```
>>> factor(x**2 + 4*x*y + 4*y**2)
(x + 2*y) ** 2
>>> factor(x**3 + y**3, extension=sqrt(-3))
(x + y) * (x + y * (1/2 - sqrt(3)*i/2)) * (x + y * (1/2 + sqrt(3)*i/2))
```

2.8.5 Gröbner bases

Buchberger's algorithm is implemented, supporting various monomial orders

```
>>> groebner([x**2 + 1, y**4*x + x**3])
GroebnerBasis([ $\begin{bmatrix} 2 & 4 \\ x^2 + 1 & y^4 - 1 \end{bmatrix}$ ], x, y, domain= $\mathbb{Z}$ , order=lex)
```

```
>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], order=grevlex)
GroebnerBasis([ $\begin{bmatrix} 4 & 3 & 2 \\ y^4 - 1 & z & x^2 + 1 \end{bmatrix}$ ], x, y, z, domain= $\mathbb{Z}$ , order=grevlex)
```

2.9 Matrices

To make a matrix in Diofant, use the [Matrix](#) (page 432) object. A matrix is constructed by providing a list of row vectors that make up the matrix.

```
>>> Matrix([[1, -1], [3, 4], [0, 2]])
 $\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 0 & 2 \end{bmatrix}$ 
```

A list of elements is considered to be a column vector.

```
>>> Matrix([1, 2, 3])
 $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ 
```

One important thing to note about Diofant matrices is that, unlike every other object in Diofant, they are mutable. This means that they can be modified in place, as we will see below. Use [ImmutableMatrix](#) (page 498) in places that require immutability, such as inside other Diofant expressions or as keys to dictionaries.

2.9.1 Indexing

Diofant matrices support subscription of matrix elements with pair of integers or [slice](#) instances. In last case, new [Matrix](#) (page 432) instances will be returned.

```
>>> M = Matrix([[1, 2, 3], [4, 5, 6]])
>>> M[0, 1]
2
>>> M[1, 1]
5
>>> M[:, 1]
 $\begin{bmatrix} 2 \\ 5 \end{bmatrix}$ 
>>> M[1, :-1]
 $\begin{bmatrix} 4 & 5 \end{bmatrix}$ 
>>> M[0, :]
 $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ 
```

(continues on next page)

(continued from previous page)

```
>>> M[:, -1]
[3]
[6]
```

It's possible to modify matrix elements.

```
>>> M[0, 0] = 0
>>> M
[0 2 3]
[4 5 6]
>>> M[1, 1:] = Matrix([[0, 0]])
>>> M
[0 2 3]
[4 0 0]
```

2.9.2 Reshape and Rearrange

To get the shape of a matrix use [shape](#) (page 469) property

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

To delete a row or column, use `del`

```
>>> del M[:, 0]
>>> M
[2 3]
[0 4]
>>> del M[1, :]
>>> M
[2 3]
```

To insert rows or columns, use methods [row_insert\(\)](#) (page 468) or [col_insert\(\)](#) (page 448).

```
>>> 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]
```

Note: You can see, that these methods will modify the Matrix **in place**. In general, as a rule, such methods will return `None`.

To swap two given rows or columns, use methods `row_swap()` (page 484) or `col_swap()` (page 482).

```
>>> M.row_swap(0, 1)
>>> M

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

>>> M.col_swap(1, 2)
>>> M

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

```

To take the transpose of a Matrix, use `T` (page 444) property.

```
>>> M.T

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

```

2.9.3 Algebraic Operations

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, N = Matrix([[1, 3], [-2, 3]]), Matrix([[0, 3], [0, 7]])
>>> M+N

$$\begin{bmatrix} 1 & 6 \\ -2 & 10 \end{bmatrix}$$

>>> M*N

$$\begin{bmatrix} 0 & 24 \\ 0 & 24 \end{bmatrix}$$

>>> 3*M

$$\begin{bmatrix} 3 & 9 \\ -6 & 9 \end{bmatrix}$$

>>> M**2

$$\begin{bmatrix} -6 & 9 \\ -5 & 12 \end{bmatrix}$$

>>> M**-1

$$\begin{bmatrix} 1/3 & -1/3 \\ 2/9 & 1/9 \end{bmatrix}$$

>>> N**-1
Traceback (most recent call last):
ValueError: Matrix det == 0; not invertible.
```

2.9.4 Special Matrices

Several constructors exist for creating common matrices. To create an identity matrix, use `eye()` (page 474) function.

```
>>> 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()` (page 473) function.

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

Similarly, function `ones()` (page 473) creates a matrix of ones.

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

To create diagonal matrices, use function `diag()` (page 474). Its arguments can be either numbers or matrices. A number is interpreted as a 1×1 matrix. The matrices are stacked diagonally.

```
>>> 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]]
```

2.9.5 Advanced Methods

To compute the determinant of a matrix, use `det()` (page 449) method.

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

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

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

To put a matrix into reduced row echelon form, use method `rref()` (page 468). It 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.

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

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

>>> _.rref()

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

```

To find the nullspace of a matrix, use method `nullspace()` (page 465). It returns a list of column vectors that span the nullspace of the matrix.

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

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

>>> _.nullspace()

$$\left[ \begin{bmatrix} -15 \\ 6 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ -1/2 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right]$$

```

To find the eigenvalues of a matrix, use method `eigenvals()` (page 451). It returns a dictionary of roots including its multiplicity (similar to the output of `roots()` (page 542) function).

```
>>> M = Matrix([[3, -2, 4, -2], [5, -2, 2, -2], [5, -2, -3, -2], [5, -2, -3, 3]])
>>> M

$$\begin{bmatrix} 3 & -2 & 4 & -2 \\ 5 & 3 & -3 & -2 \\ 5 & -2 & 2 & -2 \\ 5 & -2 & -3 & 3 \end{bmatrix}$$

```

(continues on next page)

(continued from previous page)

```
>>> 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.

Matrices can have symbolic elements.

```
>>> Matrix([[x, x + y], [y, x]])

$$\begin{bmatrix} x & x + y \\ y & x \end{bmatrix}$$

>>> _.eigenvals()

$$\left\{ x - \sqrt{y \cdot (x + y)} : 1, x + \sqrt{y \cdot (x + y)} : 1 \right\}$$

```

To find the eigenvectors of a matrix, use method `eigenvects()` (page 451).

```
>>> M.eigenvects()

$$\left( \begin{pmatrix} -2, 1, \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{pmatrix}, \begin{pmatrix} 3, 1, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{pmatrix}, \begin{pmatrix} 5, 2, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix} \end{pmatrix} \right)$$

```

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 method `diagonalize()` (page 450). It returns a tuple (P, D) , where D is diagonal and $M = PDP^{-1}$.

```
>>> M.diagonalize()

$$\left( \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix} \right)$$

>>> _[0]*_[1]*_[0]**-1 == M
True
```

If all you want is the characteristic polynomial, use method `charpoly()` (page 446). This is more efficient than `eigenvals()` (page 451) method, because sometimes symbolic roots can be expensive to calculate.

```
>>> M.charpoly(x)
PurePoly(x**4 - 11*x**3 + 29*x**2 + 35*x - 150, x, domain='ZZ')
>>> factor(_)
(x - 5)**2*(x - 3)*(x + 2)
```

To compute Jordan canonical form J for matrix M and its similarity transformation P (i.e. such that $J = PMP^{-1}$), use method `jordan_form()` (page 462).

```
>>> Matrix([[-2, 4], [1, 3]]).jordan_form()

$$\begin{pmatrix} \frac{1}{2} + \frac{\sqrt{41}}{2} & 0 \\ 0 & -\frac{\sqrt{41}}{2} + \frac{1}{2} \end{pmatrix}, \begin{pmatrix} -4 & -4 \\ -\frac{\sqrt{41}}{2} - \frac{5}{2} & -\frac{5}{2} + \frac{\sqrt{41}}{2} \\ 1 & 1 \end{pmatrix}$$

```

2.10 Expression Trees

Most generic interface to represent a mathematical expression in Diofant is a tree. Let us take the expression

```
>>> x*y + x**2

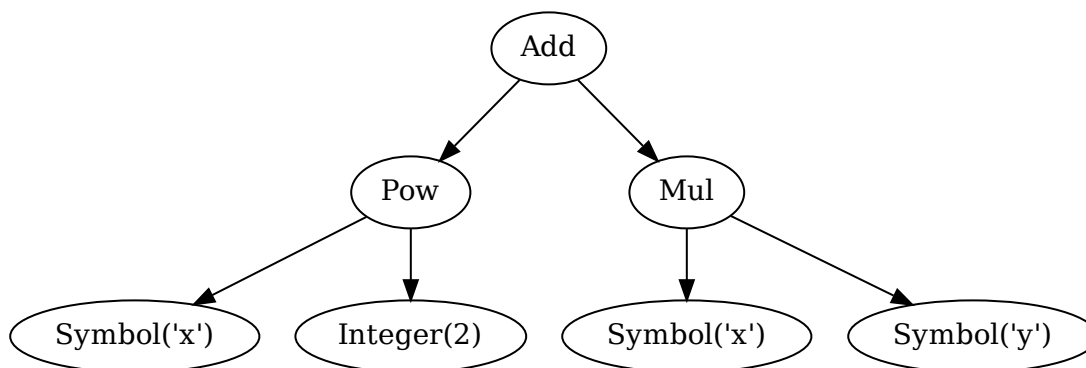
$$x^2 + x \cdot y$$

```

We can see what this expression looks like internally by using `repr()`

```
>>> repr(_)
"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



First, let's look at the leaves of this tree. We got here instances of the [Symbol](#) (page 80) class and the Diofant version of integers, instance of the [Integer](#) (page 89) class, even technically we input integer literal 2.

What about $x*y$? As we might expect, this is the multiplication of x and y . The Diofant class for multiplication is [Mul](#) (page 101)

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

Thus, we could have created the same object by writing

```
>>> Mul(x, y)
x·y
```

When we write x^{**2} , this creates a [Pow](#) (page 99) class instance

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

We could have created the same object by calling

```
>>> Pow(x, 2)
x2
```

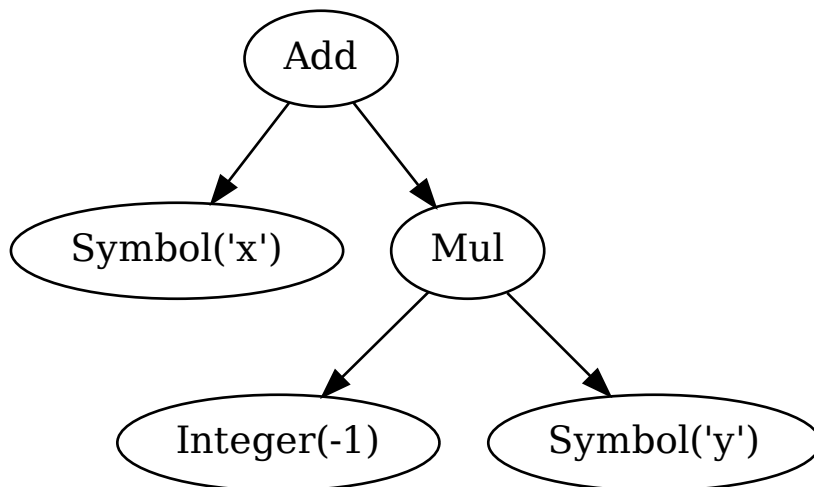
Now we get to our final expression, $x*y + x^{**2}$. This is the addition of our last two objects. The Diofant class for addition is [Add](#) (page 104), so, as you might expect, to create this object, we also could use

```
>>> Add(Pow(x, 2), Mul(x, y))
x2 + x·y
```

Note: The arguments of [Add](#) (page 104) and the commutative arguments of [Mul](#) (page 101) are stored in an order, which is independent of the order inputted.

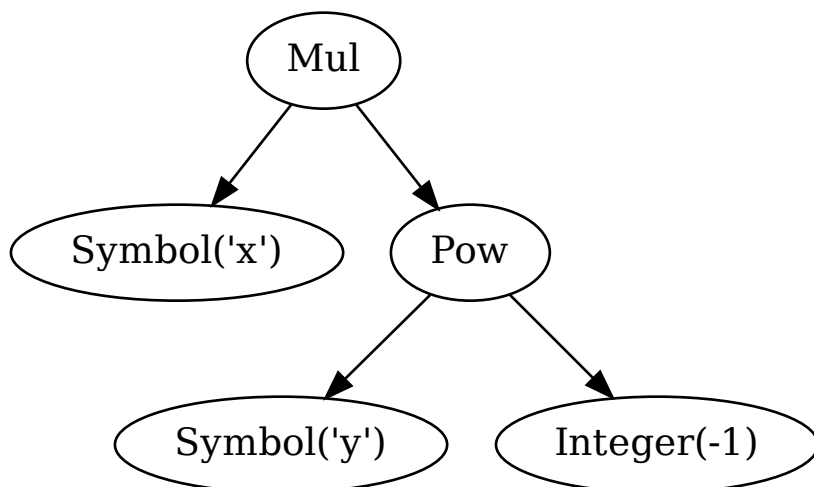
There is no subtraction class. $x - y$ is represented as $x + (-1)*y$

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



Similarly to subtraction, there is no division class

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



We see that x/y is represented as $x*y**(-1)$.

But what about $x/2$? Following the pattern of the previous example, we might expect to see `Mul(x, Pow(Integer(2), -1))`. But instead, we have

```
>>> repr(x/2)
"Mul(Rational(1, 2), Symbol('x'))"
```

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 $1/2$.

2.10.1 Walking the Tree

Let's look at how to dig our way through an expression tree, using a very generic interface — attributes *func* (page 48) and *args* (page 46).

The head of the object is encoded in the *func* (page 48) attribute

```
>>> expr = 2 + x*y
>>> expr
x*y + 2
>>> expr.func
<class 'diofant.core.add.Add'>
```

The class of an object need not be the same as the one used to create it

```
>>> Add(x, x)
2*x
>>> _.func
<class 'diofant.core.mul.Mul'>
```

Note: Diofant classes heavily use the `__new__()` class constructor, which, unlike `__init__()`, allows a different class to be returned from the constructor.

The children of a node in the tree are held in the `args` (page 46) attribute

```
>>> expr.args  
(2, x.y)
```

Note: Every expression with non-empty `args` (page 46) can be reconstructed, using

```
>>> expr.func(*expr.args)  
x.y + 2
```

Empty `args` (page 46) signal that we have hit a leaf of the expression tree

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

This interface allows us to write recursive generators that walk expression trees either in post-order or pre-order fashion

```
>>> tuple(preorder_traversal(expr))  
(x.y + 2, 2, x.y, x̄, y)  
>>> tuple(postorder_traversal(expr))  
(2, x, y, x.y, x.y + 2)
```


COMMAND-LINE USAGE

When called as a program from the command line, the following form is used:

3.1 python -m diofant

Python shell for Diofant.

This is just a normal Python shell (IPython shell if you have the IPython package installed), that adds default imports and run some initialization code.

```
usage: python -m diofant [-h] [--no-wrap-division] [-a] [--no-ipython]
                        [--unicode-identifiers] [--wrap-floats] [-V]
```

-h, --help

show this help message and exit

--no-wrap-division

Don't wrap integer divisions with Fraction

-a, --auto-symbols

Automatically create missing Symbol's

--no-ipython

Don't use IPython

--unicode-identifiers

Allow any unicode identifiers

--wrap-floats

Wrap float literals with Float

-V, --version

Print the Diofant version and exit

REFERENCE

Most of the things are already documented though in this document, that is automatically generated using Diofant's docstrings.

Tip: Because every feature of Diofant must have a test case, when you are not sure how to use something, just look into the `tests` subdirectories, find that feature and read the tests for it.

Warning: The Diofant project is in the early stage of developement and has no stable API yet.

4.1 Config

Configuration utilities.

`diofant.config.setup(key, value=None)`

Assign a value to (or reset) a configuration item.

4.2 Core

Core module. Provides the basic operations needed in diofant.

4.2.1 sympify

sympify

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

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

For example, it will convert Python ints into instance of `diofant.Rational`, floats into instances of `diofant.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 diofant
- standard numeric python types: int, long, float, Decimal
- strings (like “0.09” or “2e-19”)
- booleans, including None (will leave None unchanged)
- lists, sets or tuples containing any of the above

If the argument is already a type that Diofant 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.

```
>>> 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'"
```

Locals

The sympification happens with access to everything that is loaded by from diofant import *; anything used in a string that is not defined by that import will be converted to a symbol. In the following, the 0 is interpreted as the Order object (used with series) and it raises an error when used improperly:

```
>>> sympify('0(x)')
0(x)
>>> sympify('0 + 1')
Traceback (most recent call last):
TypeError: unbound method...
```

In order to have the 0 interpreted as a Symbol, identify it as such in the namespace dictionary. This can be done in a variety of ways; all three of the following are possibilities:

```
>>> ns = {}
>>> ns['0'] = Symbol('0') # method 1
>>> exec('from diofant.abc import 0', ns) # method 2
>>> ns.update({0: Symbol('0')}) # method 3
>>> sympify('0 + 1', locals=ns)
0 + 1
```

If you want *all* single-letter and Greek-letter variables to be symbols then you can use the clashing-symbols dictionaries that have been defined there as private variables: _clash1 (single-letter variables), _clash2 (the multi-letter Greek names) or _clash (both single and multi-letter names that are defined in abc).

```
>>> from diofant.abc import _clash1
>>> _clash1
{'E': E, 'I': I, 'N': N, 'O': 0, 'S': S}
>>> sympify('E & O', _clash1)
E & 0
```

Strict

If the option `strict` is set to `True`, only the types for which an explicit conversion has been defined are converted. In the other cases, a `SympifyError` is raised.

```
>>> print(sympify(None))
None
>>> sympify(None, strict=True)
Traceback (most recent call last):
SympifyError: SympifyError: None
```

Evaluation

If the option `evaluate` is set to `False`, then arithmetic and operators will be converted into their Diofant equivalents and the `evaluate=False` option will be added. Nested `Add` or `Mul` will be denested first. This is done via an AST transformation that replaces operators with their Diofant equivalents, so if an operand redefines any of those operations, the redefined operators will not be used.

```
>>> sympify('2**2 / 3 + 5')
19/3
>>> sympify('2**2 / 3 + 5', evaluate=False)
2**2/3 + 5
```

Sometimes autosimplification during sympification results in expressions that are very different in structure than what was entered. Below you can see how an expression reduces to `-1` by autosimplification, but does not do so when `evaluate` option is used.

```
>>> -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
>>> sympify(s, evaluate=False)
-2*((x - 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
```

Extending

To extend `sympify` to convert custom objects (not derived from `Basic`), just define a `_diofant_` method to your class. You can do that even to classes that you do not own by subclassing or adding the method at runtime.

```
>>> class MyList1:
...     def __iter__(self):
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i):
...         return list(self)[i]
...     def _diofant_(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 Diofant object, e.g. `converter[MyList] = lambda x: Matrix(x)`.

```
>>> class MyList2: # XXX Do not do this if you control the class!
...     def __iter__(self): # Use _diofant_!
...         yield 1
...         yield 2
```

(continues on next page)

(continued from previous page)

```

...     return
...     def _getitem_(self, i):
...         return list(self)[i]
>>> converter[MyList2] = lambda x: Matrix(x)
>>> sympify(MyList2())
Matrix([
[1]
[2]])

```

4.2.2 assumptions

This module contains the machinery handling assumptions.

All symbolic objects have assumption attributes that can be accessed via `.is_<assumption name>` attribute, i.e. `is_integer` (page 73). Full set of defined assumption names are accessible as `Expr.default_assumptions.rules.defined_facts` attribute.

Assumptions determine certain properties of symbolic objects and can have 3 possible values: True, False, None. True is returned if the object has the property and False is returned if it doesn't or can't (i.e. doesn't make sense):

```

>>> I.is_algebraic
True
>>> I.is_real
False
>>> I.is_prime
False

```

When the property cannot be determined (or when a method is not implemented) None will be returned, e.g. a generic symbol, `x`, may or may not be positive so a value of None is returned for `x.is_positive`.

By default, all symbolic values are in the largest set in the given context without specifying the property. For example, a symbol that has a property being integer, is also real, complex, etc.

See also:

[ImaginaryUnit](#) (page 96), [is_algebraic](#) (page 69), [is_real](#) (page 76), [is_prime](#) (page 75)

Notes

Assumption values are stored in `obj._assumptions` dictionary or are returned by getter methods (with property decorators) or are attributes of objects/classes.

class `diofant.core.assumptions.ManagedProperties(*args, **kws)`

Metaclass for classes with old-style assumptions.

class `diofant.core.assumptions.StdFactKB(facts=None)`

A FactKB specialised for the built-in rules

This is the only kind of FactKB that Basic objects should use.

`diofant.core.assumptions.as_property(fact)`

Convert a fact name to the name of the corresponding property.

`diofant.core.assumptions.check_assumptions(expr, **assumptions)`

Checks if expression *expr* satisfies all assumptions.

Parameters

- **expr** (*Expr*) – Expression to test.
- ****assumptions** (*dict*) – Keyword arguments to specify assumptions to test.

Returns

True, False or None (if can't conclude).

Examples

```
>>> check_assumptions(-5, integer=True)
True
>>> x = Symbol('x', positive=True)
>>> check_assumptions(2*x + 1, negative=True)
False
>>> check_assumptions(z, real=True) is None
True
```

`diofant.core.assumptions.make_property(fact)`

Create the automagic property corresponding to a fact.

4.2.3 cache

cacheit

`diofant.core.cache.cacheit(f, maxsize=None)`

Caching decorator.

The result of cached function must be *immutable*.

Examples

```
>>> @cacheit
... def f(a, b):
...     print(a, b)
...     return a + b
>>> f(x, y)
x y
x + y
>>> f(x, y)
x + y
```

4.2.4 basic

Base class for all the objects in Diofant.

class diofant.core.basic.Atom(*args)

A parent class for atomic things.

An atom is an expression with no subexpressions, for example Symbol, Number, Rational or Integer, but not Add, Mul, Pow.

classmethod class_key()

Nice order of classes.

doit(**hints)

Evaluate objects that are not evaluated by default.

See also:

[*Basic.doit*](#) (page 47)

sort_key(order=None)

Return a sort key.

class diofant.core.basic.Basic(*args)

Base class for all objects in Diofant.

Always use args property, when accessing parameters of some instance.

property args

Returns a tuple of arguments of 'self'.

Examples

```
>>> cot(x).args
(x,)
>>> (x*y).args
(x, y)
```

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

```
>>> e = 1 + x + 2*sin(y + I*pi)
>>> e.atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> e.atoms(Symbol)
{x, y}
```

```
>>> e.atoms(Number)
{1, 2}
```

```
>>> e.atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> e.atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that `I` (imaginary unit) and `zoo` (complex infinity) are special types of number symbols and are not part of the `NumberSymbol` class.

The type can be given implicitly, too:

```
>>> e.atoms(x)
{x, y}
```

Be careful to check your assumptions when using the implicit option since `Integer(1).is_Integer = True` but `type(Integer(1))` is `One`, a special type of diofant atom, while `type(Integer(2))` is type `Integer` and will find all integers in an expression:

```
>>> e.atoms(Integer(1))
{1}
```

```
>>> e.atoms(Integer(2))
{1, 2}
```

Finally, arguments to `atoms()` can select more than atomic atoms: any diofant type can be listed as an argument and those types of “atoms” as found in scanning the arguments of the expression recursively:

```
>>> from diofant.core.function import AppliedUndef
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

```
>>> f = Function('f')
>>> e = 1 + f(x) + 2*sin(y + I*pi)
>>> e.atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

classmethod class_key()

Nice order of classes.

copy()

Return swallow copy of self.

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.

For example, 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'.

Examples

```
>>> 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)
```

find(query)

Find all subexpressions matching a query.

property free_symbols

Return from the atoms of self those which are free symbols.

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

property func

The top-level function in an expression.

The following should hold for all objects:

```
x == x.func(*x.args)
```

Examples

```
>>> a = 2*x
>>> a.func
<class 'diofant.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

has(*patterns)

Test if any subexpression matches any of the patterns.

Parameters

***patterns** (*tuple of Expr*) - List of expressions to search for match.

Returns

bool - False if there is no match or patterns list is empty, else True.

Examples

```
>>> e = x**2 + sin(x*y)
>>> e.has(z)
False
>>> e.has(x, y, z)
True
>>> x.has()
False
```

property `is_evaluated`

Test if an expression is evaluated.

`match(pattern)`

Pattern matching.

Wild symbols match all.

Parameters

pattern (*Expr*) – An expression that may contain Wild symbols.

Returns

dict or None – If pattern match self, return a dictionary of replacement rules, such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> p = Wild('p')
>>> q = Wild('q')
>>> e = (x + y)**(x + y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> (p**q).xreplace(_)
(x + y)**(x + y)
```

See also:

[xreplace](#) (page 53), [diofant.core.symbol.Wild](#) (page 81)

`replace(query, value, exact=False)`

Replace matching subexpressions of self with value.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree in a simultaneous fashion so changes made are targeted only once. 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:

Examples

Initial setup

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

See also:

subs (page 52)

substitution of subexpressions as defined by the objects themselves.

xreplace (page 53)

exact node replacement in expr tree; also capable of using matching rules

rewrite(*args, **hints)

Rewrite functions in terms of other functions.

Rewrites expression containing applications of functions of one kind in terms of functions of different kind. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

As a pattern this function accepts a list of functions to to rewrite (instances of DefinedFunction class). As rule you can use string or a destination function instance (in this case rewrite() will use the str() function).

There is also the possibility to pass hints on how to rewrite the given expressions. For now there is only one such hint defined called 'deep'. When 'deep' is set to False it will forbid functions to rewrite their contents.

Examples

Unspecified pattern:

```
>>> sin(x).rewrite(exp)
-I*(E**(I*x) - E**(-I*x))/2
```

Pattern as a single function:

```
>>> sin(x).rewrite(sin, exp)
-I*(E**(I*x) - E**(-I*x))/2
```

Pattern as a list of functions:

```
>>> sin(x).rewrite([sin], exp)
-I*(E**(I*x) - E**(-I*x))/2
```

`sort_key(order=None)`

Return a sort key.

Examples

```
>>> sorted([Rational(1, 2), I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> [x, 1/x, 1/x**2, x**2, sqrt(x), root(x, 4), x**Rational(3, 2)]
[x, 1/x, x**(-2), x**2, sqrt(x), root(x, 4), sqrt(x)**3]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 17x, root(x, 4), sqrt(x), x, sqrt(x)**3, x**2]
```

`subs(*args, **kwargs)`

Substitutes old for new in an expression after sympifying args.

args is either:

- **one iterable argument, e.g. `foo.subs(iterable)`. The iterable may be**
 - o **an iterable container with (old, new) pairs. In this case the** replacements are processed in the order given with successive patterns possibly affecting replacements already made.
 - o **a dict or set whose key/value items correspond to old/new pairs.** In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default `sort_key`. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is `True`, the subexpressions will not be evaluated until all the substitutions have been made.

Examples

```
>>> (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
```


Examples

```
>>> (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})
E**y + x + 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:

```
>>> 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})
Traceback (most recent call last):
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

See also:

[replace \(page 49\)](#)

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

[subs \(page 52\)](#)

substitution of subexpressions as defined by the objects themselves.

class `diofant.core.basic.preorder_traversal`(*node*, *keys=None*)

Do a pre-order traversal of a tree.

This iterator recursively yields nodes that it has visited in a pre-order fashion. That is, it yields the current node then descends through the tree breadth-first to yield all of a node's children's pre-order traversal.

For an expression, the order of the traversal depends on the order of `.args`, which in many cases can be arbitrary.

Parameters

- **node** (*diofant 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.

Yields

subtree (*diofant expression*) - All of the subtrees in the tree.

Examples

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(preorder_traversal((x + y)*z, keys=True))
[z*(x + y), z, x + y, x, y]
```

skip()

Skip yielding current node's (last yielded node's) subtrees.

Examples

```
>>> pt = preorder_traversal((x+y*z)*z)
>>> for i in pt:
...     print(i)
...     if i == x + y*z:
...         pt.skip()
z*(x + y*z)
z
x + y*z
```

4.2.5 core**4.2.6 singleton**

Singleton mechanism

diofant.core.singleton.S: [SingletonRegistry](#) (page 56) = S

Alias for instance of [SingletonRegistry](#) (page 56).

class diofant.core.singleton.Singleton(*args, **kwargs)

Metaclass for singleton classes.

A singleton class has only one instance which is returned every time the class is instantiated. Additionally, this instance can be accessed through the global registry object S (page 55) as S.<class_name>.

Examples

```
>>> class MySingleton(Basic, metaclass=Singleton):
...     pass
>>> Basic() is Basic()
False
>>> MySingleton() is MySingleton()
True
>>> S.MySingleton is MySingleton()
True
```

Notes

Instance creation is delayed until the first time the value is accessed.

This metaclass is a subclass of `ManagedProperties` because that is the metaclass of many classes that need to be Singletons (Python does not allow subclasses to have a different metaclass than the superclass, except the subclass may use a subclassed metaclass).

class diofant.core.singleton.SingletonRegistry

The registry for the singleton classes.

Several classes in Diofant appear so often that they are singletonized, that is, using some metaprogramming they are made so that they can only be instantiated once (see the [diofant.core.singleton.Singleton](#) (page 55) class for details). For instance, every time you create `Integer(0)`, this will return the same instance, [diofant.core.numbers.Zero](#) (page 92).

```
>>> a = Integer(0)
>>> a is S.Zero
True
```

All singleton instances are attributes of the [S](#) (page 55) object, so `Integer(0)` can also be accessed as `S.Zero`.

Notes

For the most part, the fact that certain objects are singletonized is an implementation detail that users shouldn't need to worry about. In Diofant library code, `is` comparison is often used for performance purposes. The primary advantage of [S](#) (page 55) for end users is the convenient access to certain instances that are otherwise difficult to type, like `S.Half` (instead of `Rational(1, 2)`).

When using `is` comparison, make sure the argument is a [Basic](#) (page 46) instance. For example,

```
>>> int(0) is S.Zero
False
```

class diofant.core.singleton.SingletonWithManagedProperties(*args, **kwargs)

Metaclass for singleton classes with managed properties.

4.2.7 evaluate

diofant.core.evaluate.evaluate(x)

Control automatic evaluation.

This context managers controls whether or not all Diofant functions evaluate by default.

Note that much of Diofant expects evaluated expressions. This functionality is experimental and is unlikely to function as intended on large expressions.

Examples

```
>>> x + x
2*x
>>> with evaluate(False):
x + x
```

4.2.8 expr

4.2.9 Expr

class diofant.core.expr.Expr(*args, **kwargs)

Base class for algebraic expressions.

Everything that requires arithmetic operations to be defined should subclass this class, instead of Basic (which should be used only for argument storage and expression manipulation, i.e. pattern matching, substitutions, etc).

See also:

[diofant.core.basic.Basic](#) (page 46)

adjoint()

Compute conjugate transpose or Hermite conjugation.

See also:

[diofant.functions.elementary.complexes.adjoint](#) (page 277)

apart(x=None, **args)

See the apart function in diofant.polys.

args_cnc(cset=False, warn=True, split_1=True)

Return [commutative factors, non-commutative factors] of self.

self is treated as a Mul and the ordering of the factors is maintained. If cset is True the commutative factors will be returned in a set. If there were repeated factors (as may happen with an unevaluated Mul) then an error will be raised unless it is explicitly suppressed by setting warn to False.

Note: -1 is always separated from a Number unless split_1 is False.

```
>>> A, B = symbols('A B', commutative=0)
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], [T]]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], [T]]
>>> (-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)
[[-1, 2, x, y], [T]]
```

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], [T]]
```

as_base_exp()

Return base and exp of self.

See also:

[*diofant.core.power.Pow.as_base_exp*](#) (page 100)

as_coeff_Add(*rational=False*)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_coeff_add(deps*)**

Return the tuple (c, args) where self is written as an Add, a.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of a; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you don't know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you don't want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use self.as_independent(*deps)

```
>>> (Integer(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)

$c \cdot x^e \rightarrow c, e$ where x can be any symbolic expression.

as_coeff_mul(deps*, ***kwargs*)**

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

c should be a Rational multiplied by any 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)

```
>>> (Integer(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.

Examples

```
>>> 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 = (2*E + x*E).as_poly()
>>> p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
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)
```

See also:

coeff (page 65)

return sum of terms have a given factor

as_coeff_Add (page 58)

separate the additive constant from an expression

`as_coeff_Mul` (page 58)

separate the multiplicative constant from an expression

`as_independent` (page 61)

separate x-dependent terms/factors from others

`diofant.polys.polytools.Poly.coeff_monomial` (page 509)

efficiently find the single coefficient of a monomial in Poly

`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

```
>>> (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

```
>>> 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 Diofant expression.

Examples

```
>>> 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, Integer(1)) or (Integer(1), self) is returned.

To force the expression to be treated as an Add, use the hint `as_Add=True`

Examples

- self is an Add

```
>>> (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

```
>>> 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, E**(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

```
>>> 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

```
>>> separatevars(exp(x+y)).as_independent(x)
(E**y, E**x)
```

(continues on next page)

(continued from previous page)

```
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b = symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))
```

See also:

[diofant.simplify.simplify.separatevars](#) (page 582), [expand](#) (page 67), [diofant.core.add.Add.as_two_terms](#) (page 105), [diofant.core.mul.Mul.as_two_terms](#) (page 102), [as_coeff_add](#) (page 58), [as_coeff_mul](#) (page 58)

as_leading_term(x)

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

```
>>> (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 $\rightarrow a/b \rightarrow a, b$.

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

normal (page 76)

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

```
>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(*gens, **args)

Converts self to a polynomial or returns None.

Examples

```
>>> (x**2 + x*y).as_poly()
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> (x**2 + x*y).as_poly(x, y)
Poly(x**2 + x*y, x, y, domain='ZZ')
```

```
>>> (x**2 + sin(y)).as_poly(x, y) is None
True
```

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.

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

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> (z + t*I).as_real_imag()
(re(z) - im(t), re(t) + im(z))
```

as_terms()

Transform an expression to a list of terms.

aseries(x, n=6, bound=0, hir=False)

Returns asymptotic expansion for "self".

This is equivalent to `self.series(x, oo, n)`

Use the `hir` parameter to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

If the most rapidly varying subexpression of a given expression `f` is `f` itself, the algorithm tries to find a normalized representation of the mrv set and rewrites `f` using this normalized representation. Use the `bound` parameter to give limit on rewriting coefficients in its normalized form.

If the expansion contains an order term, it will be either $O(x^{**(-n)})$ or $O(w^{**(-n)})$ where w belongs to the most rapidly varying expression of `self`.

Examples

```
>>> e = sin(1/x + exp(-x)) - sin(1/x)
>>> e.series(x, oo)
E**(-x)*(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), x, oo))
>>> e.aseries(x, n=3, hir=True)
-E**(-2*x)*sin(1/x)/2 + E**(-x)*cos(1/x) + O(E**(-3*x), x, oo)
```

```
>>> e = exp(exp(x)/(1 - 1/x))
>>> e.aseries(x, bound=3)
E**(E**x)*E**(E**x/x**2)*E**(E**x/x)*E**(-E**x + E**x/(1 - 1/x) - E**x/x -
E**x/x**2)
>>> e.series(x, oo)
E**(E**x/(1 - 1/x))
```

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz [Gru96], p.90. It majorly uses the `mrsv` and `rewrite` sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression w of a given expression f and then expands f in a series in w . Then same thing is recursively done on the leading coefficient till we get constant coefficients.

References

- https://en.wikipedia.org/wiki/Asymptotic_expansion

cancel(*gens, **args)

See the `cancel` function in `diofant.polys`.

property 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

```
>>> Lambda(x, 2*x).canonical_variables
{x: 0_}
```

coeff(x, n=1, right=False)

Returns the coefficient from the term(s) containing x^{**n} . If n is zero then all terms independent of x will be returned.

When x is noncommutative, the coefficient to the left (default) or right of x can be returned. The keyword 'right' is ignored when x is commutative.

See also:

`diofant.core.expr.Expr.as_coefficient` (page 59), `diofant.core.expr.Expr.as_coeff_Add` (page 58), `diofant.core.expr.Expr.as_coeff_Mul` (page 58),

diofant.core.expr.Expr.as_independent (page 61), *diofant.polys.polytools.Poly.coeff_monomial* (page 509)

Examples

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
>>> (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:

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> (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 diofant.simplify.

combsimp()
See the combsimp function in diofant.simplify.

compute_leading_term(x, logx=None)
as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

conjugate()
Returns the complex conjugate of self.

See also:

[diofant.functions.elementary.complexes.conjugate](#) (page 278)

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}.

```
>>> (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.

diff(*args, **kwargs)
Alias for [diff\(\)](#) (page 125).

equals(other, failing_expression=False)
Return True if self == other, False if it doesn't, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.
If self is a Number (or complex number) that is not zero, then the result is False.
If self is a number and has not evaluated to zero, evalf will be used to test whether the expression evaluates to zero. If it does so and the result has significance (i.e. the precision is either -1, for a Rational result, or is greater than 1) then the evalf value will be used to return True or False.

expand(*deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints*)

Expand an expression using hints.

See also:

[*diofant.core.function.expand*](#) (page 129)

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.

Examples

```
>>> 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
```

See also:

[*extract_multiplicatively*](#) (page 68), [*coeff*](#) (page 65), [*as_coefficient*](#) (page 59)

extract_branch_factor(*allow_half=False*)

Try to write self as $\exp_{\text{polar}}(2\pi i n)z$ in a nice way. Return (*z*, *n*).

```
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If *allow_half* is True, also extract $\exp_{\text{polar}}(I\pi)$:

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(*c*)

Return None if it's not possible to make self in the form *c* * something in a nice way, i.e. preserving the properties of arguments of self.

```
>>> 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 diofant.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

```
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

integrate(*args, **kwargs)

See the integrate function in diofant.integrals.

invert(other, *gens, **args)

Return the multiplicative inverse of self mod other where self (and other) may be symbolic expressions).

See also:

[diofant.core.numbers.mod_inverse](#) (page 91), [diofant.polys.polytools.invert](#) (page 532)

property is_algebraic

Test if self can have only values from the set of algebraic numbers.

References

- https://en.wikipedia.org/wiki/Algebraic_number

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.

Examples

```
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

See also:

[*is_rational_function*](#) (page 75)

References

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

property `is_commutative`

Test if self commutes with any other object wrt multiplication operation.

property `is_comparable`

Test if self can be computed to a real number with precision.

Examples

```
>>> (I*exp_polar(I*pi/2)).is_comparable
True
>>> (I*exp_polar(I*pi*2)).is_comparable
False
```

property `is_complex`

Test if self can have only values from the set of complex numbers.

See also:

[*is_real*](#) (page 76)

property `is_composite`

Test if self is a positive integer that has at least one positive divisor other than 1 or the number itself.

References

- https://en.wikipedia.org/wiki/Composite_number

`is_constant(*wrt, **flags)`

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

If an expression has no free symbols then it is a constant. If there are free symbols it is possible that the expression is a constant, perhaps (but not necessarily) zero. To test such expressions, two strategies are tried:

1) numerical evaluation at two random points. If two such evaluations give two different values and the values have a precision greater than 1 then self is not constant. If the evaluations agree or could not be obtained with any precision, no decision is made. The numerical testing is done only if wrt is different than the free symbols.

2) differentiation with respect to variables in 'wrt' (or all free symbols if omitted) to see if the expression is constant or not. This will not always lead to an expression that is zero even though an expression is constant (see added test in test_expr.py). If all derivatives are zero then self is constant with respect to the given symbols.

If neither evaluation nor differentiation can prove the expression is constant, None is returned unless two numerical values happened to be the same and the flag `fail-ing_number` is True - in that case the numerical value will be returned.

If flag `simplify=False` is passed, self will not be simplified; the default is True since self should be simplified before testing.

Examples

```
>>> x.is_constant()
False
>>> Integer(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
```

property is_even

Test if self can have only values from the set of even integers.

See also:

[*is_odd*](#) (page 74)

References

- https://en.wikipedia.org/wiki/Parity_%28mathematics%29

property is_extended_real

Test if self can have only values on the extended real number line.

See also:

[*is_real*](#) (page 76)

References

- https://en.wikipedia.org/wiki/Extended_real_number_line

property is_finite

Test if self absolute value is bounded.

References

- <https://en.wikipedia.org/wiki/Finite>

is_hypergeometric(*n*)

Test if self is a hypergeometric term in *n*.

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.

See also:

[*diofant.simplify.simplify.hypersimp*](#) (page 584)

property is_imaginary

Test if self is an imaginary number.

I.e. that it can be written as a real number multiplied by the imaginary unit *I*.

References

- https://en.wikipedia.org/wiki/Imaginary_number

property is_infinite

Test if self absolute value can be arbitrarily large.

References

- `math.isfinite()`
- `numpy.isfinite`

property is_integer

Test if self can have only values from the set of integers.

property is_irrational

Test if self value cannot be represented exactly by Rational.

References

- https://en.wikipedia.org/wiki/Irrational_number

property is_negative

Test if self can have only negative values.

References

- https://en.wikipedia.org/wiki/Negative_number

property is_noninteger

Test if self can have only values from the subset of real numbers, that aren't integers.

property is_nonnegative

Test if self can have only nonnegative values.

See also:

[*is_negative*](#) (page 73)

References

- https://en.wikipedia.org/wiki/Negative_number

property is_nonpositive

Test if self can have only nonpositive values.

property is_nonzero

Test if self is nonzero.

See also:

[*is_zero*](#) (page 76)

property is_number

Returns True if 'self' has no free symbols.

It will be faster than `if not self.free_symbols`, however, since `is_number` will fail as soon as it hits a free symbol.

Examples

```
>>> 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
```

property is_odd

Test if self can have only values from the set of odd integers.

See also:

[*is_even*](#) (page 71)

References

- https://en.wikipedia.org/wiki/Parity_%28mathematics%29

property is_polar

Test if self can have values from the Riemann surface of the logarithm.

See also:

[*diofant.functions.elementary.complexes.polar_lift*](#) (page 278), [*diofant.functions.elementary.complexes.principal_branch*](#) (page 279), [*diofant.functions.elementary.exponential.exp_polar*](#) (page 295)

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

```
>>> ((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.

```
>>> 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*](#) (page 75)

property `is_positive`

Test if self can have only positive values.

property `is_prime`

Test if self is a natural number greater than 1 that has no positive divisors other than 1 and itself.

References

- https://en.wikipedia.org/wiki/Prime_number

property `is_rational`

Test if self can have only values from the set of rationals.

`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

```
>>> (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.

```
>>> 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*](#) (page 69)

property `is_real`

Test if self can have only values from the set of real numbers.

See also:

[*is_complex*](#) (page 70)

References

- https://en.wikipedia.org/wiki/Real_number

property `is_transcendental`

Test if self can have only values from the set of transcendental numbers.

References

- https://en.wikipedia.org/wiki/Transcendental_number

property `is_zero`

Test if self is zero.

See also:

[*is_nonzero*](#) (page 73)

`limit(x, xlim, dir=-1)`

Compute limit $x \rightarrow xlim$.

normal()

Canonicalize ratio, i.e. return numerator if denominator is 1.

nseries(x, n=6, logx=None)

Calculate “n” terms of series in x around 0

This calculates n terms of series 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.

Parameters

- **x** (*Symbol*) – variable for series expansion (positive and finite symbol)
- **n** (*Integer, optional*) – number of terms to calculate. Default is 6.
- **logx** (*Symbol, optional*) – This can be used to replace any $\log(x)$ in the returned series with a symbolic value to avoid evaluating $\log(x)$ at 0.

See also:

[series](#) (page 79)

Examples

```
>>> sin(x).nseries(x)
x - x**3/6 + x**5/120 + 0(x**7)
>>> log(x + 1).nseries(x, 5)
x - x**2/2 + x**3/3 - x**4/4 + 0(x**5)
```

Handling of the logx parameter — in the following example the expansion fails since sin does not have an asymptotic expansion at -oo (the limit of $\log(x)$ as x approaches 0).

```
>>> e = sin(log(x))
>>> e.nseries(x)
Traceback (most recent call last):
PoleError:
>>> logx = Symbol('logx')
>>> e.nseries(x, logx=logx)
sin(logx)
```

Notes

This method call the helper method `_eval_nseries`. Such methods should be implemented in subclasses.

The series expansion code is an important part of the gruntz algorithm for determining limits. `_eval_nseries` has to return a generalized power series with coefficients in $C(\log(x), \log)$:

```
c_0*x**e_0 + ... (finitely many terms)
```

where e_i are numbers (not necessarily integers) and c_i involve only numbers, the function `log`, and `log(x)`. (This also means it must not contain `log(x(1 + p))`, this *has* to be expanded to `log(x) + log(1 + p)` if `p.is_positive`.)

nsimplify(*constants*=[], *tolerance*=None, *full*=False)

See the `nsimplify` function in `diofant.simplify`.

powsimp(***args*)

See the `powsimp` function in `diofant.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

```
>>> (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, (x/2 + 3)*(6*x + 2))
```

radsimp(***kwargs*)

See the `radsimp` function in `diofant.simplify`.

ratsimp()

See the `ratsimp` function in `diofant.simplify`.

remove0()

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.

Examples

```
>>> Float(10.5).round()
11.
>>> pi.round()
3.
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6.0 + 3.0*I
```

The `round` method has a chopping effect:

```
>>> (2*pi + I/10).round()
6.
>>> (pi/10 + 2*I).round()
2.0*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

Notes

Do not confuse the Python builtin function, `round`, with the Diofant 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(Integer(123), -2), Number)
False
>>> isinstance(Integer(123).round(-2), Number)
True
>>> isinstance((3*I).round(), Mul)
True
>>> isinstance((1 + 3*I).round(), Add)
True
```

series(*x=None*, *x0=0*, *n=6*, *dir=None*, *logx=None*)

Series expansion of “self” around $x = x_0$ yielding either terms of the series one by one (the lazy series given when $n=None$), else all the terms at once when $n \neq None$.

Returns the series expansion of “self” around the point $x = x_0$ with respect to x up to $O((x - x_0)^{n+1})$ (default n is 6).

If $x=None$ and `self` is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

```
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, x, 1)
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(E**y) - x*sin(E**y) + O(x**2)
```

If $n=None$ then a generator of the series terms will be returned.

```
>>> term = cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For $dir=-1$ (default) the series is calculated from the right and for $dir=+1$ the series from the left. For infinite x_0 (∞ or $-\infty$), the dir argument is determined from the direction of the infinity (i.e. $dir=+1$ for ∞). For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir=-1)
x
>>> abs(x).series(dir=+1)
-x
```

For rational expressions this method may return original expression.

```
>>> (1/x).series(x, n=8)
1/x
```

simplify(*ratio=1.7, measure=None*)

See the simplify function in diofant.simplify.

sort_key(*order=None*)

Return a sort key.

taylor_term(*n, x, *previous_terms*)

General method for the taylor term.

This method is slow, because it differentiates n-times. Subclasses can redefine it to make it faster by using the “previous_terms”.

together(**args, **kwargs*)

See the together function in diofant.polys.

transpose()

Transpose self.

See also:

[*diofant.functions.elementary.complexes.transpose*](#) (page 280)

trigsimp(***args*)

See the trigsimp function in diofant.simplify.

4.2.10 AtomicExpr

class diofant.core.expr.**AtomicExpr**(**args, **kwargs*)

A parent class for object which are both atoms and Exprs.

For example: Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

4.2.11 symbol

Symbol

class diofant.core.symbol.**Symbol**(*name, **assumptions*)

Symbol is a placeholder for atomic symbolic expression.

It has a name and a set of assumptions.

Parameters

- **name** (*str*) – The name for Symbol.
- ****assumptions** (*dict*) – Keyword arguments to specify assumptions for Symbol. Default assumption is commutative=True.

Examples

```
>>> a, b = symbols('a b')
>>> bool(a*b == b*a)
True
```

You can override default assumptions:

```
>>> A, B = symbols('A B', commutative=False)
>>> bool(A*B != B*A)
True
>>> bool(A*B*2 == 2*A*B) is True # multiplication by scalars is commutative
True
```

See also:

[diofant.core.assumptions](#) (page 44), [Dummy](#) (page 82), [Wild](#) (page 81)

Wild

class `diofant.core.symbol.Wild(name, exclude=(), properties=(), **assumptions)`

A Wild symbol matches anything, whatever is not explicitly excluded.

Examples

```
>>> a = Wild('a')
>>> x.match(a)
{a_: x}
>>> pi.match(a)
{a_: pi}
>>> (3*x**2).match(a*x)
{a_: 3*x}
>>> cos(x).match(a)
{a_: cos(x)}
>>> b = Wild('b', exclude=[x])
>>> (3*x**2).match(b*x)
>>> b.match(a)
{a_: b_}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A_}
```

Notes

When using Wild, be sure to use the `exclude` keyword to make the pattern more precise. Without the `exclude` pattern, you may get matches that are technically correct, but not what you wanted. For example, using the above without `exclude`:

```
>>> a, b = symbols('a b', cls=Wild)
>>> (2 + 3*y).match(a*x + b*y)
{a_: 2/x, b_: 3}
```

This is technically correct, because $(2/x)*x + 3*y == 2 + 3*y$, but you probably wanted it to not match at all. The issue is that you really didn't want `a` and `b` to include `x` and `y`, and the `exclude` parameter lets you specify exactly this. With the `exclude` parameter, the pattern will not match.

```
>>> a = Wild('a', exclude=[x, y])
>>> b = Wild('b', exclude=[x, y])
>>> (2 + 3*y).match(a*x + b*y)
```

Exclude also helps remove ambiguity from matches.

```
>>> E = 2*x**3*y*z
>>> a, b = symbols('a b', cls=Wild)
>>> E.match(a*b)
{a_: 2*y*z, b_: x**3}
>>> a = Wild('a', exclude=[x, y])
>>> E.match(a*b)
{a_: z, b_: 2*x**3*y}
>>> a = Wild('a', exclude=[x, y, z])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

See also:

[Symbol](#) (page 80)

Dummy

class diofant.core.symbol.Dummy(*name=None, **assumptions*)

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

```
>>> bool(Dummy('x') == Dummy('x')) is 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.

See also:

[Symbol](#) (page 80)

classmethod class_key()

Nice order of classes.

sort_key(*order=None*)

Return a sort key.

symbols

diofant.core.symbol.symbols(*names, **args*)

Transform strings into instances of [Symbol](#) (page 80) class.

[symbols\(\)](#) (page 82) 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:

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

The type of output is dependent on the properties of input arguments:

```
>>> symbols('x')
x
>>> symbols('x,')
(x,)
>>> symbols('x,y')
(x, y)
>>> symbols(('a', 'b', 'c'))
(a, b, c)
>>> symbols(['a', 'b', 'c'])
[a, b, c]
>>> symbols({'a', 'b', 'c'})
{a, b, c}
```

If an iterable container is needed for a single symbol, set the `seq` argument to `True` or terminate the symbol name with a comma:

```
>>> symbols('x', seq=True)
(x,)
```

To reduce typing, range syntax is supported to create indexed symbols. Ranges are indicated by a colon and the type of range is determined by the character to the right of the colon. If the character is a digit then all contiguous digits to the left are taken as the nonnegative starting value (or 0 if there is no digit left of the colon) and all contiguous digits to the right are taken as 1 greater than the ending value:

```
>>> symbols('x:10')
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
>>> symbols('x5:10')
(x5, x6, x7, x8, x9)
>>> symbols('x5(:2)')
(x50, x51)
>>> symbols('x5:10 y:5')
(x5, x6, x7, x8, x9, y0, y1, y2, y3, y4)
>>> symbols(('x5:10', 'y:5'))
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))
```

If the character to the right of the colon is a letter, then the single letter to the left (or 'a' if there is none) is taken as the start and all characters in the lexicographic range *through* the letter to the right are used as the range:

```
>>> symbols('x:z')
(x, y, z)
>>> symbols('x:c') # null range
()
>>> symbols('x(:c)')
(xa, xb, xc)
>>> symbols(':c')
(a, b, c)
>>> symbols('a:d, x:z')
(a, b, c, d, x, y, z)
>>> symbols(('a:d', 'x:z'))
((a, b, c, d), (x, y, z))
```

Multiple ranges are supported; contiguous numerical ranges should be separated by parentheses to disambiguate the ending number of one range from the starting number of the next:

```
>>> symbols('x:2(1:3)')
(x01, x02, x11, x12)
>>> symbols(':3:2') # parsing is from left to right
(00, 01, 10, 11, 20, 21)
```

Only one pair of parentheses surrounding ranges are removed, so to include parentheses around ranges, double them. And to include spaces, commas, or colons, escape them with a backslash:

```
>>> symbols('x((a:b))')
(x(a), x(b))
>>> symbols(r'x(:1\,:2)') # or '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 82) 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 'diofant.core.function.UndefinedFunction'>
```

var

`diofant.core.symbol.var(names, **args)`

Create symbols and inject them into the global namespace.

This calls `symbols()` (page 82) with the same arguments and puts the results into the *global* namespace. It's recommended not to use `var()` (page 84) in library code, where `symbols()` (page 82) has to be used.

Examples

```
>>> 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 also:

[`symbols`](#) (page 82)

4.2.12 numbers

Number

class diofant.core.numbers.Number(*obj)

Represents any kind of number in diofant.

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.

as_coeff_Add(*rational=False*)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_coeff_add(*deps)

Return the tuple (c, args) where self is written as an Add.

See also:

[diofant.core.expr.Expr.as_coeff_add](#) (page 58)

as_coeff_mul(*deps, **kwargs)

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

See also:

[diofant.core.expr.Expr.as_coeff_mul](#) (page 58)

classmethod class_key()

Nice order of classes.

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*.

sort_key(*order=None*)

Return a sort key.

Float

class diofant.core.numbers.Float(num, dps=None)

Represent a floating-point number of arbitrary precision.

Examples

```
>>> Float(3.5)
3.5000000000000000
>>> Float(3)
3.
```

Creating Floats from strings (and Python int type) will give a minimum precision of 15 digits, but the precision will automatically increase to capture all digits entered.

```
>>> Float(1)
1.
>>> Float(10**20)
100000000000000000000.
>>> Float('1e20')
1.e+20
```

However, *floating-point* numbers (Python float types) retain only 15 digits of precision:

```
>>> Float(1e20)
1.000000000000000e+20
>>> Float('1.23456789123456789')
1.23456789123457
```

It may be preferable to enter high-precision decimal numbers as strings:

```
>>> Float('1.23456789123456789')
1.23456789123456789
```

The desired number of digits can also be specified:

```
>>> Float('1e-3', 3)
0.00100
>>> Float(100, 4)
100.0
```

Float can automatically count significant figures if decimal precision argument is omitted. (Auto-counting is only allowed for strings and ints).

```
>>> Float('12e-3')
0.012
>>> Float(3)
3.
>>> Float('60.e2') # 2 digits significant
6.0e+3
>>> Float('6000.') # 4 digits significant
6000
>>> Float('600e-2') # 3 digits significant
6.00
```

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, you can change the precision of Float, 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:

```
>>> Float(approx, 5)
0.099609
```

Please note that you can't increase precision with evalf:

```
>>> approx.evalf(5)
Traceback (most recent call last):
PrecisionExhausted: ...
```

By contrast, 0.125 is exact in binary (as it is in base 10) and so it can be passed to Float constructor to obtain an arbitrary precision with matching accuracy:

```
>>> Float(exact, 5)
0.12500
>>> Float(exact, 20)
0.125000000000000000000000
```

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)
0.299999999999999999998890
```

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)
0.300000000000000000000000
```

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 mpmath.libmp import to_man_exp
...     from diofant import Mul, Pow
...     m, e = to_man_exp(f.mpf_, signed=True)
...     v = Mul(int(m), Pow(2, int(e), evaluate=False), evaluate=False)
...     print(f'{v} at prec={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
```

ceiling()

Compute ceiling of self.

epsilon_eq(other, epsilon='1e-15')

Test approximate equality.

floor()

Compute floor of self.

Rational

class diofant.core.numbers.**Rational**(*p*, *q*=1)

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

Examples

```
>>> 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
```

The conversion of floats to expressions or simple fractions can be handled with `nsimplify`:

```
>>> 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:

```
>>> r = Rational(3, 4)
>>> r
3/4
>>> r.numerator
3
>>> r.denominator
4
```

Note that these properties return integers (not Diofant Integers) so some care is needed when using them in expressions:

```
>>> r.numerator/r.denominator
0.75
```

See also:

[`diofant.core.sympify.sympify`](#) (page 41), [`diofant.simplify.simplify.nsimplify`](#) (page 585)

as_content_primitive(*radical=False*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> Rational(-3, 2).as_content_primitive()
(3/2, -1)
```

See also:

[`diofant.core.expr.Expr.as_content_primitive`](#) (page 60)

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.

gcd(*other*)

Compute GCD of *self* and *other*.

lcm(*other*)

Compute LCM of *self* and *other*.

limit_denominator(*max_denominator=1000000*)

Closest Rational to self with denominator at most *max_denominator*.

```
>>> Rational('3.141592653589793').limit_denominator(10)
22/7
>>> Rational('3.141592653589793').limit_denominator(100)
311/99
```

Integer

class `diofant.core.numbers.Integer`(*i*)

Represents integer numbers.

property `is_composite`

Test if self is a positive integer that has at least one positive divisor other than 1 or the number itself.

References

- https://en.wikipedia.org/wiki/Composite_number

property `is_even`

Test if self can have only values from the set of even integers.

See also:

[*is_odd*](#) (page 90)

References

- https://en.wikipedia.org/wiki/Parity_%28mathematics%29

property `is_imaginary`

Test if self is an imaginary number.

I.e. that it can be written as a real number multiplied by the imaginary unit I .

References

- https://en.wikipedia.org/wiki/Imaginary_number

property `is_nonzero`

Test if self is nonzero.

See also:

[*is_zero*](#) (page 91)

property `is_odd`

Test if self can have only values from the set of odd integers.

See also:

[*is_even*](#) (page 90)

References

- https://en.wikipedia.org/wiki/Parity_%28mathematics%29

property `is_prime`

Test if self is a natural number greater than 1 that has no positive divisors other than 1 and itself.

References

- https://en.wikipedia.org/wiki/Prime_number

property `is_zero`

Test if self is zero.

See also:

[*is_nonzero*](#) (page 90)

NumberSymbol

class `diofant.core.numbers.NumberSymbol`

Base class for symbolic numbers.

method `approximation_interval(number_cls)`

Return an interval with `number_cls` endpoints that contains the value of `NumberSymbol`. If not implemented, then return `None`.

`mod_inverse`

`diofant.core.numbers.mod_inverse(a, m)`

Return the number `c` such that, $(a * c) \% m == 1$ where `c` has the same sign as `a`. If no such value exists, a `ValueError` is raised.

Examples

Suppose we wish to find multiplicative inverse `x` of 3 modulo 11. This is the same as finding `x` such that $3 * x = 1 \pmod{11}$. One value of `x` that satisfies this congruence is 4. Because $3 * 4 = 12$ and $12 = 1 \pmod{11}$. This is the value return by `mod_inverse`:

```
>>> mod_inverse(3, 11)
4
>>> mod_inverse(-3, 11)
-4
```

When there is a common factor between the numerators of `a` and `m` the inverse does not exist:

```
>>> mod_inverse(2, 4)
Traceback (most recent call last):
ValueError: inverse of 2 mod 4 does not exist
```

```
>>> mod_inverse(Integer(2)/7, Integer(5)/2)
7/2
```

References

- https://en.wikipedia.org/wiki/Modular_multiplicative_inverse
- https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

integer_digits

`diofant.core.numbers.integer_digits(n, b)`

Gives a list of the base *b* digits in the integer *n*.

Zero

class `diofant.core.numbers.Zero(*args, **kwargs)`

The number zero.

Zero is a singleton, and can be accessed by `S.Zero`

Examples

```
>>> Integer(0) is S.Zero
True
>>> 1/S.Zero
zoo
```

References

- <https://en.wikipedia.org/wiki/Zero>

One

class `diofant.core.numbers.One(*args, **kwargs)`

The number one.

One is a singleton, and can be accessed by `S.One`.

Examples

```
>>> Integer(1) is S.One
True
```


References

- https://en.wikipedia.org/wiki/1_%28number%29

NegativeOne

class diofant.core.numbers.NegativeOne(*args, **kwargs)

The number negative one.

NegativeOne is a singleton, and can be accessed by S.NegativeOne.

Examples

```
>>> Integer(-1) is S.NegativeOne
True
```

See also:

[One](#) (page 92)

References

- https://en.wikipedia.org/wiki/%E2%88%921_%28number%29

Half

class diofant.core.numbers.Half(*args, **kwargs)

The rational number 1/2.

Half is a singleton, and can be accessed by S.Half.

Examples

```
>>> Rational(1, 2) is S.Half
True
```

References

- https://en.wikipedia.org/wiki/One_half

NaN

class diofant.core.numbers.NaN(*args, **kwargs)

Not a Number.

This serves as a place holder for numeric values that are indeterminate. Most operations on NaN, produce another NaN. Most indeterminate forms, such as $0/0$ or $\infty - \infty$ produce NaN. Two exceptions are $0^{**}0$ and $\infty^{**}0$, which all produce 1 (this is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`. Differences are noted below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the initially counter-intuitive results with `Eq` and `==` in the examples below.

NaN is not comparable so inequalities raise a `TypeError`. This is in contrast with floating point nan where all inequalities are false.

NaN is a singleton, and can be accessed by `nan`.

Examples

```
>>> nan is nan
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan)    # mathematical equality
False
>>> nan == nan      # structural equality
True
```

References

- <https://en.wikipedia.org/wiki/NaN>

Infinity

class diofant.core.numbers.Infinity(*args, **kwargs)

Positive infinite quantity.

In real analysis the symbol ∞ denotes an unbounded limit: $x \rightarrow \infty$ means that x grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled $+\infty$ and $-\infty$ can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by `oo`, or can be imported as `oo`.

Examples

```
>>> 1 + oo
oo
>>> 42/oo
0
>>> limit(exp(x), x, oo)
oo
```

See also:

NegativeInfinity (page 95), *NaN* (page 94)

References

- <https://en.wikipedia.org/wiki/Infinity>

NegativeInfinity

class diofant.core.numbers.**NegativeInfinity**(*args, **kwargs)

Negative infinite quantity.

NegativeInfinity is a singleton, and can be accessed by -oo.

See also:

Infinity (page 94)

ComplexInfinity

class diofant.core.numbers.**ComplexInfinity**(*args, **kwargs)

Complex infinity.

In complex analysis the symbol ∞ , called “complex infinity”, represents a quantity with infinite magnitude, but undetermined complex phase.

ComplexInfinity is a singleton, and can be accessed by as zoo.

Examples

```
>>> zoo + 42
zoo
>>> 42/zoo
0
>>> zoo + zoo
nan
>>> zoo*zoo
zoo
```

See also:

Infinity (page 94)

Exp1

class diofant.core.numbers.**Exp1**(*args, **kwargs)

The e constant.

The transcendental number $e = 2.718281828\dots$ is the base of the natural logarithm and of the exponential function, $e = \exp(1)$. Sometimes called Euler's number or Napier's constant.

Exp1 is a singleton, and can be imported as E.

Examples

```
>>> E is exp(1)
True
>>> log(E)
1
```

References

- https://en.wikipedia.org/wiki/E_%28mathematical_constant%29

ImaginaryUnit

class diofant.core.numbers.**ImaginaryUnit**(*args, **kwargs)

The imaginary unit, $i = \sqrt{-1}$.

I is a singleton, and can be imported as I.

Examples

```
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```

References

- https://en.wikipedia.org/wiki/Imaginary_unit

Pi

class diofant.core.numbers.**Pi**(*args, **kwargs)

The π constant.

The transcendental number $\pi = 3.141592654\dots$ represents the ratio of a circle's circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

Pi is a singleton, and can be imported as pi.

Examples

```

>>> pi > 3
true
>>> pi.is_irrational
True
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)

```

References

- <https://en.wikipedia.org/wiki/Pi>

EulerGamma

class diofant.core.numbers.**EulerGamma**(*args, **kwargs)

The Euler-Mascheroni constant.

$\gamma = 0.5772157\dots$ (also called Euler's constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right)$$

Examples

```

>>> EulerGamma.is_irrational
>>> EulerGamma > 0
true
>>> EulerGamma > 1
false

```

References

- https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant

Catalan

class diofant.core.numbers.**Catalan**(*args, **kwargs)

Catalan's constant.

$K = 0.91596559\dots$ is given by the infinite series

$$K = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

Examples

```
>>> Catalan.is_irrational
>>> Catalan > 0
true
>>> Catalan > 1
false
```

References

- https://en.wikipedia.org/wiki/Catalan%27s_constant

GoldenRatio

class diofant.core.numbers.**GoldenRatio**(*args, **kwargs)

The golden ratio, ϕ .

$\phi = \frac{1+\sqrt{5}}{2}$ is algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum.

Examples

```
>>> GoldenRatio > 1
true
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> GoldenRatio.is_irrational
True
```

References

- https://en.wikipedia.org/wiki/Golden_ratio

4.2.13 power

Pow

class diofant.core.power.Pow(*b, e, evaluate=None*)

Defines the expression x^y as “ x raised to a power y ”.

For complex numbers x and y , Pow gives the principal value of $\exp(y * \log(x))$.

Singleton definitions involving (0, 1, -1, oo, -oo, I, -I):

expr	value	reason
$z^{**}0$	1	Although arguments over $0^{**}0$ exist, see [2].
$z^{**}1$	z	
$(-\infty)^{**}(-1)$	0	
$(-1)^{**}-1$	-1	
$0^{**}-1$	zoo	This is not strictly true, as $0^{**}-1$ may be undefined, but is convenient in some contexts where the base is assumed to be positive.
$1^{**}-1$	1	
$\infty^{**}-1$	0	
$0^{**}\infty$	0	Because for all complex numbers z near 0, $z^{**}\infty \rightarrow 0$.
$0^{**}-\infty$	zoo	This is not strictly true, as $0^{**}\infty$ may be oscillating between positive and negative values or rotating in the complex plane. It is convenient, however, when the base is positive.
$1^{**}\infty$	nan	Because there are various cases where $\lim(x(t), t)=1$, $\lim(y(t), t)=\infty$ (or $-\infty$), but $\lim(x(t)^{**}y(t), t) \neq 1$. See [3].
$1^{**}-\infty$	nan	
$z^{**}zoo$	nan	No limit for $z^{**}t$ for $t \rightarrow zoo$.
$(-1)^{**}\infty$	nan	Because of oscillations in the limit.
$(-1)^{**}(-\infty)$	nan	
$\infty^{**}\infty$	oo	
$\infty^{**}-\infty$	0	
$(-\infty)^{**}\infty$	nan	
$(-\infty)^{**}-\infty$	nan	
$\infty^{**}I$	nan	$\infty^{**}e$ could probably be best thought of as the limit of $x^{**}e$ for real x as x tends to ∞ . If e is I , then the limit does not exist and nan is used to indicate that.
$(-\infty)^{**}I$	nan	
$\infty^{**}(1+I)$	zoo	If the real part of e is positive, then the limit of $\text{abs}(x^{**}e)$ is ∞ . So the limit value is zoo.
$(-\infty)^{**}(1+I)$	zoo	
$\infty^{**}(-1+I)$	0	If the real part of e is negative, then the limit is 0.
$(-\infty)^{**}(-1+I)$	0	

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754 conventions. This helps us avoid extra test-case code in the calculation of limits.

See also:

diofant.core.numbers.Infinity (page 94), *diofant.core.numbers.NaN* (page 94)

References

- <https://en.wikipedia.org/wiki/Exponentiation>
- https://en.wikipedia.org/wiki/Zero_to_the_power_of_zero
- https://en.wikipedia.org/wiki/Indeterminate_forms

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

```
>>> p = Pow(Rational(1, 2), 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

```
>>> 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)))
```

```
>>> ((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 also:

[*diofant.core.expr.Expr.as_content_primitive*](#) (page 60)

as_real_imag(deep=True, **hints)

Returns real and imaginary parts of self

See also:

[*diofant.core.expr.Expr.as_real_imag*](#) (page 64)

property base

Returns base of the power expression.

classmethod class_key()

Nice order of classes.

property exp

Returns exponent of the power expression.

integer_nthroot

diofant.core.power.integer_nthroot(y, n)

Return a tuple containing $x = \text{floor}(y^{1/n})$ and a boolean indicating whether the result is exact (that is, whether $x^n == y$).

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

4.2.14 mul

Mul

class diofant.core.mul.Mul(*args, **options)

Symbolic multiplication class.

as_base_exp()

Return base and exp of self.

See also:

[*diofant.core.expr.Expr.as_base_exp*](#) (page 57)

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

as_coeff_mul(*deps, **kwargs)

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

See also:

[*diofant.core.expr.Expr.as_coeff_mul*](#) (page 58)

as_content_primitive(radical=False)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(-sqrt(2) + 1))
```

See also:

[*diofant.core.expr.Expr.as_content_primitive*](#) (page 60)

as_ordered_factors(order=None)

Transform an expression into an ordered list of factors.

Examples

```
>>> (2*x*y*sin(x)*cos(x)).as_ordered_factors()
[2, x, y, sin(x), cos(x)]
```

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power.

See also:

[*diofant.core.expr.Expr.as_powers_dict*](#) (page 64)

as_real_imag(deep=True, **hints)

Returns real and imaginary parts of self

See also:

[*diofant.core.expr.Expr.as_real_imag*](#) (page 64)

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use `self.args[0]`;
- if you want to process the arguments of the tail then use `self.as_coef_mul()` which gives the head and a tuple containing the arguments of the tail when treated as a Mul.
- if you want the coefficient when self is treated as an Add then use `self.as_coeff_add()[0]`

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

classmethod class_key()

Nice order of classes.

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 diofant as `Mul(Mul(a, b), c)`. This can have undesirable consequences.
 - Sometimes terms are not combined as one would like: {c.f. [sympy/sympy#4596](#)}

```
>>> 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. [sympy/sympy#5728](#)}

```
>>> a = sqrt(x*sqrt(y))
>>> a**3
sqrt(x*sqrt(y))**3
>>> Mul(a, a, a)
sqrt(x*sqrt(y))**3
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> .subs({a.base: z}).subs({z: a.base})
sqrt(x*sqrt(y))**3
```

- 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 101) 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. [sympy/sympy#5706](#)}

This consideration is moot if the cache is turned off.

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.

4.2.15 add

Add

class diofant.core.add.Add(*args, **options)

Symbolic addition class.

as_coeff_Add(rational=False)

Efficiently extract the coefficient of a summation.

as_coeff_add(*deps)

Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.

Examples

```
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

as_coefficients_dict()

Return a dictionary mapping terms to their Rational coefficient.

Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```
>>> (3*x + x*y + 4).as_coefficients_dict()
{1: 4, x: 3, x*y: 1}
>>> _[y]
0
>>> (3*y*x).as_coefficients_dict()
{x*y: 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

```
>>> (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 also:

[*diofant.core.expr.Expr.as_content_primitive*](#) (page 60)

as_real_imag(deep=True, **hints)

Returns a tuple representing a complex number.

Examples

```
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use `self.args[0]`;
- if you want to process the arguments of the tail then use `self.as_coef_add()` which gives the head and a tuple containing the arguments of the tail when treated as an Add.
- if you want the coefficient when self is treated as a Mul then use `self.as_coeff_mul()[0]`

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

classmethod class_key()

Nice order of classes.

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.

See also:

[*diofant.core.mul.Mul.flatten*](#) (page 103)

get0()

Returns the additive $O(\cdot)$ symbol.

See also:

[*diofant.core.expr.Expr.get0*](#) (page 69)

primitive()

Return $(R, \text{self}/R)$ where R is the Rational GCD of self .

R is collected only from the leading coefficient of each term.

Examples

```
>>> (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:

[*diofant.polys.polytools.primitive*](#) (page 533)

remove0()

Removes the additive $O(\cdot)$ symbol.

See also:

[*diofant.core.expr.Expr.remove0*](#) (page 78)

4.2.16 mod

Mod

class `diofant.core.mod.Mod(p, q)`

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

```
>>> x**2 % y
x**2%y
>>> _.subs({x: 5, y: 6})
1
```

4.2.17 relational

Rel

diofant.core.relational.**Rel**
alias of *Relational* (page 108)

Eq

diofant.core.relational.**Eq**
alias of *Equality* (page 109)

Ne

diofant.core.relational.**Ne**
alias of *Unequality* (page 115)

Lt

diofant.core.relational.**Lt**
alias of *StrictLessThan* (page 118)

Le

diofant.core.relational.**Le**
alias of *LessThan* (page 112)

Gt

diofant.core.relational.**Gt**
alias of *StrictGreaterThan* (page 115)

Ge

diofant.core.relational.Ge

alias of [GreaterThan](#) (page 110)

Relational

class diofant.core.relational.Relational(*lhs, rhs=0, rop=None, **assumptions*)

Base class for all relation types.

Subclasses of Relational should generally be instantiated directly, but Relational can be instantiated with a valid *rop* value to dispatch to the appropriate subclass.

Parameters

rop (*str or None*) – Indicates what subclass to instantiate. Valid values can be found in the keys of Relational.ValidRelationalOperator.

Examples

```
>>> Rel(y, x+x**2, '==')
Eq(y, x**2 + x)
```

as_set()

Rewrites univariate inequality in terms of real sets

Examples

```
>>> (x > 0).as_set()
(0, oo]
>>> Eq(x, 0).as_set()
{0}
```

property canonical

Return a canonical form of the relational.

The rules for the canonical form, in order of decreasing priority are:

- 1) Number on right if left is not a Number;
- 2) Symbol on the left;
- 3) Gt/Ge changed to Lt/Le;
- 4) Lt/Le are unchanged;
- 5) Eq and Ne get ordered args.

equals(*other, failing_expression=False*)

Return True if the sides of the relationship are mathematically identical and the type of relationship is the same. If failing_expression is True, return the expression whose truth value was unknown.

property lhs

The left-hand side of the relation.

property reversed

Return the relationship with sides (and sign) reversed.

Examples

```
>>> Eq(x, 1)
Eq(x, 1)
>>> .reversed
Eq(1, x)
>>> x < 1
x < 1
>>> .reversed
1 > x
```

property rhs

The right-hand side of the relation.

Equality

class diofant.core.relational.**Equality**(lhs, rhs=0, **options)

An equal relation between two objects.

Represents that two objects are equal. If they can be easily shown to be definitively equal (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained as an unevaluated Equality object. Use the `simplify` function on this object for more nontrivial evaluation of the equality relation.

As usual, the keyword argument `evaluate=False` can be used to prevent any evaluation.

Examples

```
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
false
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> .doit()
false
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(E**x, sinh(x) + cosh(x))
>>> simplify(_)
true
```

See also:

[diofant.logic.boolalg.Equivalent](#) (page 422)

for representing equality between two boolean expressions

Notes

This class is not the same as the `==` operator. The `==` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an `evalEq` method, it can be used in place of the default algorithm. If `lhs.evalEq(rhs)` or `rhs.evalEq(lhs)` returns anything other than `None`, that return value will be substituted for the Equality. If `None` is returned by `evalEq`, an Equality object will be created as usual.

GreaterThan

class diofant.core.relational.GreaterThan(*lhs*, *rhs*=0, ***options*)

Class representations of inequalities.

The `*Than` classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$lhs \geq rhs$

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

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

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math equivalent
GreaterThan(<i>lhs</i> , <i>rhs</i>)	$lhs \geq rhs$
LessThan(<i>lhs</i> , <i>rhs</i>)	$lhs <= rhs$
StrictGreaterThan(<i>lhs</i> , <i>rhs</i>)	$lhs > rhs$
StrictLessThan(<i>lhs</i> , <i>rhs</i>)	$lhs < rhs$

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:

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> f'{e.gts} >= {e.lts} is the same as {e.lts} <= {e.gts}'
'x >= 1 is the same as 1 <= x'
```

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
```

```
>>> print(f'Ge(x, 2)}\n{Gt(x, 2)}\n{Le(x, 2)}\n{Lt(x, 2)}')
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (\geq , $>$, \leq , $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> e2 = x >= 2
>>> print(e2)
x >= 2
>>> print(f'e1: {e1}, e2: {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:

```
>>> print(f'Rel(x, 1, '>=')}\n{Relational(x, 1, '>=')}\n{GreaterThan(x, 1)}')
x >= 1
x >= 1
x >= 1
```

```
>>> print(f'Rel(x, 1, '>')}\n{Relational(x, 1, '>')}\n{StrictGreaterThan(x, 1)}')
x > 1
x > 1
x > 1
```

```
>>> print(f'Rel(x, 1, '<=')}\n{Relational(x, 1, '<=')}\n{LessThan(x, 1)}')
x <= 1
x <= 1
x <= 1
```

```
>>> print(f'Rel(x, 1, '<')}\n{Relational(x, 1, '<')}\n{StrictLessThan(x, 1)}')
x < 1
x < 1
x < 1
```

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 Diofant object Python will try the reflective operation, $(x > 1)$. Unfortunately, there is no way available to Diofant 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\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 = Integer(1) > x
>>> e2 = Integer(1) >= x
>>> e3 = Integer(1) < x
>>> e4 = Integer(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print('%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:

```
>>> x < y < z
Traceback (most recent call last):
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python, to create a chained inequality, the only method currently available is to make use of And:

```
>>> And(x < y, y < z)
(x < y) & (y < z)
```

LessThan

class diofant.core.relational.**LessThan**(lhs, rhs=0, **options)

Class representations of inequalities.

The *Than classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs \geq 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:

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> f'{e.gts} >= {e.lts} is the same as {e.lts} <= {e.gts}'
'x >= 1 is the same as 1 <= x'
```

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
```

```
>>> print(f'{Ge(x, 2)}\n{Gt(x, 2)}\n{Le(x, 2)}\n{Lt(x, 2)}')
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(f'e1: {e1}, e2: {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:

```
>>> print(f"Rel(x, 1, '>=')\n{Relational(x, 1, '>=')}\n{GreaterThan(x, 1)}")
x >= 1
x >= 1
x >= 1
```

```
>>> print(f"Rel(x, 1, '>')\n{Relational(x, 1, '>')}\n{StrictGreaterThan(x, 1)}")
x > 1
x > 1
x > 1
```

```
>>> print(f"Rel(x, 1, '<=')\n{Relational(x, 1, '<=')}\n{LessThan(x, 1)}")
x <= 1
x <= 1
x <= 1
```

```
>>> print(f"Rel(x, 1, '<')\n{Relational(x, 1, '<')}\n{StrictLessThan(x, 1)}")
x < 1
x < 1
x < 1
```

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 Diofant object Python will try the reflective operation, $(x > 1)$. Unfortunately, there is no way available to Diofant 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\n'*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
x > 1      x >= 1
x < 1      x <= 1
1 > x      1 >= x
1 < x      1 <= x
```

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 = Integer(1) > x
>>> e2 = Integer(1) >= x
>>> e3 = Integer(1) < x
>>> e4 = Integer(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print('%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:

```
>>> x < y < z
Traceback (most recent call last):
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python, to create a chained inequality, the only method currently available is to make use of And:

```
>>> And(x < y, y < z)
(x < y) & (y < z)
```

Unequality

class diofant.core.relational.**Unequality**(lhs, rhs=0, **options)

An unequal relation between two objects.

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to False; if definitively unequal, this will reduce to True. Otherwise, the relation is maintained as an Unequality object.

Examples

```
>>> Ne(y, x+x**2)
Ne(y, x**2 + x)
```

See also:

[Equality](#) (page 109)

Notes

This class is not the same as the `!=` operator. The `!=` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of Equality. As such, it uses the same algorithms, including any available *evalEq* methods.

StrictGreaterThan

class diofant.core.relational.**StrictGreaterThan**(lhs, rhs=0, **options)

Class representations of inequalities.

The `*Than` classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$lhs \geq rhs$

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:

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> f'{e.gts} >= {e.lts} is the same as {e.lts} <= {e.gts}'
'x >= 1 is the same as 1 <= x'
```

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
```

```
>>> print(f'{Ge(x, 2)}\n{Gt(x, 2)}\n{Le(x, 2)}\n{Lt(x, 2)}')
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(f'e1: {e1}, e2: {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:


```
>>> print(f"Rel(x, 1, '>=')\n{Relational(x, 1, '>=')}\n{GreaterThan(x, 1)}")
x >= 1
x >= 1
x >= 1
```

```
>>> print(f"Rel(x, 1, '>')\n{Relational(x, 1, '>')}\n{StrictGreaterThan(x, 1)}")
x > 1
x > 1
x > 1
```

```
>>> print(f"Rel(x, 1, '<=')\n{Relational(x, 1, '<=')}\n{LessThan(x, 1)}")
x <= 1
x <= 1
x <= 1
```

```
>>> print(f"Rel(x, 1, '<')\n{Relational(x, 1, '<')}\n{StrictLessThan(x, 1)}")
x < 1
x < 1
x < 1
```

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 Diofant object Python will try the reflective operation, $(x > 1)$. Unfortunately, there is no way available to Diofant 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\n'*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
x > 1      x >= 1
x < 1      x <= 1
1 > x      1 >= x
1 < x      1 <= x
```

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 = Integer(1) > x
>>> e2 = Integer(1) >= x
>>> e3 = Integer(1) < x
>>> e4 = Integer(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print('%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:

```
>>> x < y < z
Traceback (most recent call last):
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python, to create a chained inequality, the only method currently available is to make use of And:

```
>>> And(x < y, y < z)
(x < y) & (y < z)
```

StrictLessThan

class diofant.core.relational.**StrictLessThan**(lhs, rhs=0, **options)

Class representations of inequalities.

The *Than classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs \geq rhs

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

Class Name	Symbol
GreaterThan	(\geq)
LessThan	(\leq)
StrictGreaterThan	($>$)
StrictLessThan	($<$)

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	lhs \geq rhs
LessThan(lhs, rhs)	lhs \leq 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:

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> f{e.gts} >= {e.lts} is the same as {e.lts} <= {e.gts}'
'x >= 1 is the same as 1 <= x'
```

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
```

```
>>> print(f'{Ge(x, 2)}\n{Gt(x, 2)}\n{Le(x, 2)}\n{Lt(x, 2)}')
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (\geq , $>$, \leq , $<$) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> e2 = x >= 2
>>> print(e2)
x >= 2
>>> print(f'e1: {e1}, e2: {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:

```
>>> print(f'{Rel(x, 1, '>=')} \n {Relational(x, 1, '>=')} \n {GreaterThan(x, 1)}')
x >= 1
x >= 1
x >= 1
```

```
>>> print(f'{Rel(x, 1, '>')} \n {Relational(x, 1, '>')} \n {StrictGreaterThan(x, 1)}')
x > 1
x > 1
x > 1
```

```
>>> print(f'{Rel(x, 1, '<=')} \n {Relational(x, 1, '<=')} \n {LessThan(x, 1)}')
x <= 1
x <= 1
x <= 1
```

```
>>> print(f'{Rel(x, 1, '<')} \n {Relational(x, 1, '<')} \n {StrictLessThan(x, 1)}')
x < 1
x < 1
x < 1
```

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 Diofant object Python will try the reflective operation, $(x > 1)$. Unfortunately, there is no way available to Diofant 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\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 = Integer(1) > x
>>> e2 = Integer(1) >= x
>>> e3 = Integer(1) < x
>>> e4 = Integer(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print('%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:

```
>>> x < y < z
Traceback (most recent call last):
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python, to create a chained inequality, the only method currently available is to make use of And:

```
>>> And(x < y, y < z)
(x < y) & (y < z)
```

4.2.18 multidimensional

vectorize

class diofant.core.multidimensional.**vectorize**(*mdargs)

Generalizes a function taking scalars to accept multidimensional arguments.

Examples

```
>>> @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), g(x, y)], [x, y])
[[Derivative(f(x, y), x), Derivative(f(x, y), y)],
 [Derivative(g(x, y), x), Derivative(g(x, y), y)]]
```

4.2.19 function

Lambda

class diofant.core.function.Lambda(*variables, expr, **kwargs*)

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:

```
>>> f = Lambda(x, x**2)
>>> f(4)
16
```

For multivariate functions, use:

```
>>> 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
```

property **expr**

The return value of the function.

property **free_symbols**

Return from the atoms of self those which are free symbols.

See also:

[diofant.core.basic.Basic.free_symbols](#) (page 48)

property **variables**

The variables used in the internal representation of the function.

WildFunction

class diofant.core.function.**WildFunction**(*args)

A WildFunction function matches any function (with its arguments).

Examples

```
>>> F = WildFunction('F')
>>> F.nargs
Naturals0()
>>> x.match(F)
>>> F.match(F)
{F_: F}
>>> f(x).match(F)
{F_: f(x)}
>>> cos(x).match(F)
{F_: cos(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a given number of arguments, set nargs to the desired value at instantiation:

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

To match functions with a range of arguments, set nargs to a tuple containing the desired number of arguments, e.g. if nargs = (1, 2) then functions with 1 or 2 arguments will be matched.

```
>>> F = WildFunction('F', nargs=(1, 2))
>>> F.nargs
{1, 2}
>>> f(x).match(F)
{F_: f(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
>>> f(x, y, 1).match(F)
```

Derivative

class diofant.core.function.**Derivative**(expr, *args, **assumptions)

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`.

```
>>> 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:

```
>>> (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 diofant.abc import 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(1 - sin(x)**2)
>>> 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 Diofant, because `expr.subs({Function: Symbol})` is well-defined: just structurally replace the function everywhere it appears in the expression.

This is the same notational convenience used in the Euler-Lagrange method when one

says $F(t, f(t), f'(t)).diff(f(t))$. What is actually meant is that the expression in question is represented by some $F(t, u, v)$ at $u = f(t)$ and $v = f'(t)$, and $F(t, f(t), f'(t)).diff(f(t))$ simply means $F(t, u, v).diff(u)$ at $u = f(t)$.

We do not allow derivatives to be taken with respect to expressions where this is not so well defined. For example, we do not allow `expr.diff(x*y)` because there are multiple ways of structurally defining where $x*y$ appears in an expression, some of which may surprise the reader (for example, a very strict definition would have that $(x*y*z).diff(x*y) == 0$).

```
>>> (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 Diofant is defined using unevaluated Subs objects:

```
>>> 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:

```
>>> diff(f(x), x).diff(f(x))
0
```

The same is true for derivatives of different orders:

```
>>> diff(f(x), (x, 2)).diff(diff(f(x), (x, 1)))
0
>>> diff(f(x), (x, 1)).diff(diff(f(x), (x, 2)))
0
```

Note, any class can allow derivatives to be taken with respect to itself.

Examples

Some basic examples:

```
>>> 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(hints)**

Evaluate objects that are not evaluated by default.

See also:

[*diofant.core.basic.Basic.doit*](#) (page 47)

doit_numerically(z0)

Evaluate the derivative at z numerically.

When we can represent derivatives at a point, this should be folded into the normal evalf. For now, we need a special method.

property expr

Return expression.

property free_symbols

Return from the atoms of self those which are free symbols.

See also:

[*diofant.core.basic.Basic.free_symbols*](#) (page 48)

property variables

Return tuple of symbols, wrt derivative is taken.

diff

`diofant.core.function.diff(f, *args, **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`.

Examples

```
>>> 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 'diofant.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.

References

- https://reference.wolfram.com/legacy/v5_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html

FunctionClass

class `diofant.core.function.FunctionClass(*args, **kwargs)`

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

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

property `nargs`

Return a set of the allowed number of arguments for the function.

Examples

If the function can take any number of arguments, the set of whole numbers is returned:

```
>>> Function('f').nargs
Naturals0()
```

If the function was initialized to accept one or more arguments, a corresponding set will be returned:

```
>>> Function('f', nargs=1).nargs
{1}
>>> Function('f', nargs=(2, 1)).nargs
{1, 2}
```

The undefined function, after application, also has the `nargs` attribute; the actual number of arguments is always available by checking the `args` attribute:

```
>>> f(1).nargs
Naturals0()
>>> len(f(1).args)
1
```

Function

class `diofant.core.function.Function(*args)`

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:

```
>>> g = 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 MyFunc that represents a mathematical function *MyFunc*. Suppose that it is well known, that *MyFunc*(0) is 1 and *MyFunc* at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that *MyFunc*(x) is real exactly when x is real. Here is an implementation that honours those requirements:

```
>>> class MyFunc(Function):
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x == 0:
...                 return Integer(1)
...             elif x is oo:
...                 return Integer(0)
...         def _eval_is_real(self):
...             return self.args[0].is_real
>>> MyFunc(0) + sin(0)
1
>>> MyFunc(oo)
0
>>> MyFunc(3.54).evalf() # Not yet implemented for MyFunc.
MyFunc(3.54)
>>> MyFunc(1).is_real
False
```

In order for MyFunc to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then nargs must be defined, e.g. if MyFunc can take one or two arguments then,

```
>>> class MyFunc(Function):
...     nargs = (1, 2)
```

classmethod class_key()

Nice order of classes.

fdiff(argindex=1)

Returns the first derivative of the function.

Note: Not all functions are the same

Diofant 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:

```
>>> f = Function('f')
>>> f(2) + f(x)
f(2) + f(x)
```

If you want to see which functions appear in an expression you can use the `atoms` method:

```
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function)
{f(x), cos(x)}
```

If you just want the function you defined, not Diofant functions, the thing to search for is `AppliedUndef`:

```
>>> from diofant.core.function import AppliedUndef
>>> e.atoms(AppliedUndef)
{f(x)}
```

Subs

class `diofant.core.function.Subs`(*expr*, **args*, ***assumptions*)

Represents unevaluated substitutions of an expression.

`Subs` receives at least 2 arguments: an expression, a pair of old and new expression to substitute or several such pairs.

`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()`).

Examples

```
>>> 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)
```

```
>>> Subs(f(x)*sin(y) + z, (x, 0), (y, 1))
Subs(z + f(x)*sin(y), (x, 0), (y, 1))
>>> .doit()
z + f(0)*sin(1)
```

doit(***hints*)

Evaluate objects that are not evaluated by default.

See also:

[`diofant.core.basic.Basic.doit`](#) (page 47)

evalf(*dps=15, **options*)

Evaluate the given formula to an accuracy of *dps* decimal digits.

See also:

[*diofant.core.evalf.EvalfMixin.evalf*](#) (page 138)

property expr

The expression on which the substitution operates.

property free_symbols

Return from the atoms of self those which are free symbols.

See also:

[*diofant.core.basic.Basic.free_symbols*](#) (page 48)

property point

The values for which the variables are to be substituted.

property variables

The variables to be evaluated.

expand

`diofant.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.

Parameters

- **basic** (*boolean, optional*) - This 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.
- **deep** (*boolean, optional*) - 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.
- **mul** (*boolean, optional*) - Distributes multiplication over addition (```):

```
>>> (y*(x + z)).expand(mul=True)
x*y + y*z
```

- **multinomial** (*boolean, optional*) - Expand $(x + y + \dots)^n$ where *n* is a positive integer.

```
>>> ((x + y + z)**2).expand(multinomial=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

- **power_exp** (*boolean, optional*) - Expand addition in exponents into multiplied bases.

```
>>> exp(x + y).expand(power_exp=True)
E**x*E**y
>>> (2**(x + y)).expand(power_exp=True)
2**x*2**y
```

- **power_base** (*boolean, optional*) - 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, Diofant performs it automatically:

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

- **log** (*boolean, optional*) - 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:

```
>>> 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)
```

- **complex** (*boolean, optional*) - 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** (*boolean : optional*) - Expand other functions.

```
>>> gamma(x + 1).expand(func=True)
x*gamma(x)
```

- **trig** (*boolean, optional*) - 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.

- **force** (*boolean, optional*) – If the force hint is used, assumptions about variables will be ignored in making the expansion.

Notes

- You can shut off unwanted methods:

```
>>> (exp(x + y)*(x + y)).expand()
E**x*E**y*x + E**x*E**y*y
>>> (exp(x + y)*(x + y)).expand(power_exp=False)
E**x*(x + y)*x + E**x*(x + y)*y
>>> (exp(x + y)*(x + y)).expand(mul=False)
E**x*E**y*(x + y)
```

- Use deep=False to only expand on the top level:

```
>>> exp(x + exp(x + y)).expand()
E**x*E**(E**x*E**y)
>>> exp(x + exp(x + y)).expand(deep=False)
E**(E**(x + y))*E**x
```

- 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:

```
>>> 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
```

- 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.

Examples

```
>>> 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)
```

See also:

[expand_log](#) (page 135), [expand_mul](#) (page 134), [expand_multinomial](#) (page 136), [expand_complex](#) (page 136), [expand_trig](#) (page 135), [expand_power_base](#) (page 137), [expand_power_exp](#) (page 136), [expand_func](#) (page 135), [diofant.simplify.hyperexpand.hyperexpand](#) (page 603)

References

- <https://mathworld.wolfram.com/Multiple-AngleFormulas.html>

PoleError

class diofant.core.function.PoleError

Raised when an expansion pole is encountered.

count_ops

diofant.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

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 diofant 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

diofant.core.function.expand_mul(expr, deep=True)

Wrapper around expand that only uses the mul hint. See the expand docstring for more information.

Examples

```
>>> x, y = symbols('x y', positive=True)
>>> expand_mul(exp(x+y)*(x+y)*log(x*y**2))
E**(x + y)*x*log(x*y**2) + E**(x + y)*y*log(x*y**2)
```

expand_log

diofant.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

```
>>> x, y = symbols('x y', positive=True)
>>> expand_log(exp(x+y)*(x+y)*log(x*y**2))
E**(x + y)*(x + y)*(log(x) + 2*log(y))
```

expand_func

diofant.core.function.**expand_func**(*expr*, *deep=True*)

Wrapper around `expand` that only uses the `func` hint. See the `expand` docstring for more information.

Examples

```
>>> expand_func(gamma(x + 2))
x*(x + 1)*gamma(x)
```

expand_trig

diofant.core.function.**expand_trig**(*expr*, *deep=True*)

Wrapper around `expand` that only uses the `trig` hint. See the `expand` docstring for more information.

Examples

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

expand_complex

diofant.core.function.**expand_complex**(*expr*, *deep*=True)

Wrapper around `expand` that only uses the `complex` hint. See the `expand` docstring for more information.

Examples

```
>>> expand_complex(exp(z))
E**re(z)*I*sin(im(z)) + E**re(z)*cos(im(z))
>>> expand_complex(sqrt(I))
sqrt(2)/2 + sqrt(2)*I/2
```

See also:

[*diofant.core.expr.Expr.as_real_imag*](#) (page 64)

expand_multinomial

diofant.core.function.**expand_multinomial**(*expr*, *deep*=True)

Wrapper around `expand` that only uses the `multinomial` hint. See the `expand` docstring for more information.

Examples

```
>>> x, y = symbols('x y', positive=True)
>>> expand_multinomial((x + exp(x + 1))**2)
2*E**(x + 1)*x + E**(2*x + 2) + x**2
```

expand_power_exp

diofant.core.function.**expand_power_exp**(*expr*, *deep*=True)

Wrapper around `expand` that only uses the `power_exp` hint.

Examples

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

See also:

[*expand*](#) (page 129)

expand_power_base

diofant.core.function.expand_power_base(expr, deep=True, force=False)

Wrapper around expand that only uses the power_base hint.

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.

```
>>> (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*(E**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
```

See also:

[expand](#) (page 129)

nfloat

`diofant.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

```
>>> 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
```

4.2.20 evalf

`class diofant.core.evalf.EvalfMixin`

Mixin class adding evalf capability.

`evalf(dps=15, subs=None, maxn=110, chop=False, strict=True, quad=None)`

Evaluate the given formula to an accuracy of `dps` decimal digits. Optional keyword arguments:

subs=<dict>

Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn=<integer>

Allow a maximum temporary working precision of `maxn` digits (default=110)

chop=<bool>

Replace tiny real or imaginary parts in subresults by exact zeros (default=False)

strict=<bool>

Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available `maxprec` (default=True)

quad=<str>

Choose algorithm for numerical quadrature. By default, `tanh-sinh` quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

PrecisionExhausted

`class diofant.core.evalf.PrecisionExhausted`

Raised when precision is exhausted.

N

`diofant.core.evalf.N(x, dps=15, **options)`

Calls `x.evalf(dps, **options)`.

Examples

```
>>> Sum(1/k**k, (k, 1, oo))
Sum(k**(-k), (k, 1, oo))
>>> N(_, 4)
1.291
```

See also:

[`diofant.core.evalf.EvalfMixin.evalf`](#) (page 138)

4.2.21 containers

Tuple

class `diofant.core.containers.Tuple(*args)`

Wrapper around the builtin tuple object

The Tuple is a subclass of Basic, so that it works well in the Diofant framework. The wrapped tuple is available as `self.args`, but you can also access elements or slices with `[:]` syntax.

```
>>> Tuple(a, b, c)[1:]
(b, c)
>>> Tuple(a, b, c).subs({a: d})
(d, b, c)
```

index(*value*, *start=None*, *stop=None*)

Return first index of value.

Raises `ValueError` if the value is not present.

tuple_count(*value*)

`T.count(value)` -> int - return number of occurrences of value.

Dict

class `diofant.core.containers.Dict(*args)`

Wrapper around the builtin dict object

The Dict is a subclass of Basic, so that it works well in the Diofant 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.

```
>>> D = Dict({1: 'one', 2: 'two'})
>>> for key in D:
...     if key == 1:
...         print(f'{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
```

property args

Returns a tuple of arguments of 'self'.

See also:

[*diofant.core.basic.Basic.args*](#) (page 46)

get(key, default=None)

Return the value for key if key is in the dictionary, else default.

items()

Returns a set-like object providing a view on Dict's items.

keys()

Returns a set-like object providing a view on Dict's keys.

values()

Returns a set-like object providing a view on Dict's values.

4.2.22 compatibility

Reimplementations of constructs introduced in later versions of Python than we support. Also some functions that are needed Diofant-wide and are located here for easy import.

diofant.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

```
>>> 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
```


4.2.23 exprtools

gcd_terms

`diofant.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.

Examples

```
>>> 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.

See also:

[`factor_terms`](#) (page 141), [`diofant.polys.polytools.terms_gcd`](#) (page 536)

factor_terms

`diofant.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.

Examples

```
>>> 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)
```

See also:

[`gcd_terms`](#) (page 141), [`diofant.polys.polytools.terms_gcd`](#) (page 536)

4.3 Combinatorics

4.3.1 Partitions

class `diofant.combinatorics.partitions.Partition(*partition)`

This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.

See also:

[`diofant.utilities.iterables.partitions`](#) (page 740), [`diofant.utilities.iterables.multiset_partitions`](#) (page 736)

property 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

```
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
({3}, {4}, {5}, {1, 2})
>>> .RGS
(0, 0, 1, 2, 3)
```

classmethod from_rgs(*rgs, elements*)

Creates a set partition from a restricted growth string.

The indices given in *rgs* are assumed to be the index of the element as given in *elements* *as provided* (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in *rgs* an error will be raised.

Examples

```
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
{{c}, {a, d}, {b, e}}
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
{{e}, {a, c}, {b, d}}
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
{{2}, {1, 4}, {3, 5}}
```

property partition

Return partition as a sorted list of lists.

Examples

```
>>> Partition([1], [2, 3]).partition
[[1], [2, 3]]
```

property rank

Gets the rank of a partition.

Examples

```
>>> 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

```
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key)
>>> l
[{{1, 2}}, {{1}}, {2}}, {{1, x}}, {{3, 4}}, {{0, 1, 2, 3}}]
```

class diofant.combinatorics.partitions.**IntegerPartition**(*partition*, *integer=None*)

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:

[*diofant.utilities.iterables.partitions*](#) (page 740), [*diofant.utilities.iterables.multiset_partitions*](#) (page 736)

References

- https://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

```
>>> 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

```
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
#####
#
#
```

property conjugate

Computes the conjugate partition of itself.

Examples

```
>>> 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

```
>>> 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

```
>>> p = IntegerPartition([4])
>>> print(p.prev_lex())
[3, 1]
>>> p.partition > p.prev_lex().partition
True
```

diofant.combinatorics.partitions.random_integer_partition(*n, seed=None*)

Generates a random integer partition summing to *n* as a list of reverse-sorted integers.

Examples

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]
```

`diofant.combinatorics.partitions.RGS_generalized(m)`

Computes the $m + 1$ generalized unrestricted growth strings and returns them as rows in matrix.

Examples

```
>>> 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]])
```

`diofant.combinatorics.partitions.RGS_enum(m)`

`RGS_enum` computes the total number of restricted growth strings possible for a superset of size m .

Examples

```
>>> RGS_enum(4)
15
>>> RGS_enum(5)
52
>>> RGS_enum(6)
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```
>>> a = Partition(list(range(4)))
>>> s = set()
>>> for i in range(20):
...     s.add(a)
...     a += 1
>>> assert len(s) == 15
```

`diofant.combinatorics.partitions.RGS_unrank(rank, m)`

Gives the unranked restricted growth string for a given superset size.

Examples

```
>>> RGS_unrank(14, 4)
[0, 1, 2, 3]
>>> RGS_unrank(0, 4)
[0, 0, 0, 0]
```

`diofant.combinatorics.partitions.RGS_rank(rgs)`

Computes the rank of a restricted growth string.

Examples

```
>>> RGS_rank([0, 1, 2, 1, 3])
42
>>> RGS_rank(RGS_unrank(4, 7))
4
```

4.3.2 Permutations

class `diofant.combinatorics.permutations.Permutation(*args, **kwargs)`

A permutation, alternatively known as an ‘arrangement number’ or ‘ordering’ is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement. 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 Diofant) 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.

```
>>> Permutation.print_cyclic = False
```

Notes

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 $\text{range}(n)$, where n is the size of p , it is sufficient to represent the permutation by the second line, referred to as the “array form” of the permutation. This is entered in brackets as the argument to the `Permutation` class:

```
>>> p = Permutation([0, 2, 1])
>>> p
Permutation([0, 2, 1])
```

Given i in $\text{range}(p.\text{size})$, the permutation maps i to i^p

```
>>> [i ^ p for i in range(p.size)]
[0, 2, 1]
```

The composite of two permutations $p*q$ means first apply p , then q , so $i^{(p*q)} = (i^p)^q$ which is i^{p^q} according to Python precedence rules:

```
>>> q = Permutation([2, 1, 0])
>>> [i ^ p ^ q for i in range(3)]
[2, 0, 1]
>>> [i ^ (p*q) for i in range(3)]
[2, 0, 1]
```

One can use also the notation $p(i) = i^p$, but then the composition rule is $(p*q)(i) = q(p(i))$, not $p(q(i))$:

```
>>> [(p*q)(i) for i in range(p.size)]
[2, 0, 1]
>>> [q(p(i)) for i in range(p.size)]
[2, 0, 1]
>>> [p(q(i)) for i in range(p.size)]
[1, 2, 0]
```

Disjoint Cycle Notation

In disjoint cycle notation, only the elements that have shifted are indicated. In the above case, the 2 and 1 switched places. This can be entered in two ways:

```
>>> Permutation(1, 2) == Permutation([[1, 2]]) == p
True
```

Only the relative ordering of elements in a cycle matter:

```
>>> Permutation(1, 2, 3) == Permutation(2, 3, 1) == Permutation(3, 1, 2)
True
```

The disjoint cycle notation is convenient when representing permutations that have several cycles in them:

```
>>> Permutation(1, 2)(3, 5) == Permutation([[1, 2], [3, 5]])
True
```

It also provides some economy in entry when computing products of permutations that are written in disjoint cycle notation:

```
>>> Permutation(1, 2)(1, 3)(2, 3)
Permutation([0, 3, 2, 1])
>>> _ == Permutation([[1, 2]])*Permutation([[1, 3]])*Permutation([[2, 3]])
True
```

Entering a singleton in a permutation is a way to indicate the size of the permutation. The size keyword can also be used.

Array-form entry:

```
>>> Permutation([[1, 2], [9]])
Permutation([0, 2, 1], size=10)
>>> Permutation([[1, 2]], size=10)
Permutation([0, 2, 1], size=10)
```

Cyclic-form entry:

```
>>> Permutation(1, 2, size=10)
Permutation([0, 2, 1], size=10)
>>> Permutation(9)(1, 2)
Permutation([0, 2, 1], size=10)
```


Caution: no singleton containing an element larger than the largest in any previous cycle can be entered. This is an important difference in how `Permutation` and `Cycle` handle the `__call__` syntax. A singleton argument at the start of a `Permutation` performs instantiation of the `Permutation` and is permitted:

```
>>> Permutation(5)
Permutation([], size=6)
```

A singleton entered after instantiation is a call to the permutation – a function call – and if the argument is out of range it will trigger an error. For this reason, it is better to start the cycle with the singleton:

The following fails because there is no element 3:

```
>>> Permutation(1, 2)(3)
Traceback (most recent call last):
  IndexError: list index out of range
```

This is ok: only the call to an out of range singleton is prohibited; otherwise the permutation autosizes:

```
>>> Permutation(3)(1, 2)
Permutation([0, 2, 1, 3])
>>> Permutation(1, 2)(3, 4) == Permutation(3, 4)(1, 2)
True
```

Equality testing

The array forms must be the same in order for permutations to be equal:

```
>>> Permutation([1, 0, 2, 3]) == Permutation([1, 0])
False
```

Identity Permutation

The identity permutation is a permutation in which no element is out of place. It can be entered in a variety of ways. All the following create an identity permutation of size 4:

```
>>> I = Permutation([0, 1, 2, 3])
>>> all(p == I for p in [Permutation(3), Permutation(range(4)),
...                      Permutation([], size=4), Permutation(size=4)])
True
```

Watch out for entering the range *inside* a set of brackets (which is cycle notation):

```
>>> I == Permutation([range(4)])
False
```

Permutation Printing

There are a few things to note about how Permutations are printed.

1) If you prefer one form (array or cycle) over another, you can set that with the `print_cyclic` flag.

```
>>> Permutation(1, 2)(4, 5)(3, 4)
Permutation([0, 2, 1, 4, 5, 3])
>>> p = _
```

```
>>> Permutation.print_cyclic = True
>>> p
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
>>> {i: q(i) for i in range(q.size)} # showing the bijection
{0: 5, 1: 2, 2: 3, 3: 4, 4: 1, 5: 0}
```

The full cyclic form (including singletons) can be obtained:

```
>>> p.full_cyclic_form
[[0, 1], [2], [3]]
```

Any permutation can be factored into transpositions of pairs of elements:

```
>>> Permutation([[1, 2], [3, 4, 5]]).transpositions()
[(1, 2), (3, 5), (3, 4)]
>>> Permutation.rmul(*[Permutation([ti], size=6) for ti in _]).cyclic_form
[[1, 2], [3, 4, 5]]
```

The number of permutations on a set of n elements is given by $n!$ and is called the cardinality.

```
>>> p.size
4
>>> p.cardinality
24
```

A given permutation has a rank among all the possible permutations of the same elements, but what that rank is depends on how the permutations are enumerated. (There

are a number of different methods of doing so.) The lexicographic rank is given by the `rank` method and this rank is used to increment a permutation with addition/subtraction:

```
>>> p.rank()
6
>>> p + 1
Permutation([1, 0, 3, 2])
>>> p.next_lex()
Permutation([1, 0, 3, 2])
>>> _.rank()
7
>>> p.unrank_lex(p.size, rank=7)
Permutation([1, 0, 3, 2])
```

The product of two permutations p and q is defined as their composition as functions, $(p*q)(i) = q(p(i))$.

```
>>> 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])
```

See also:

[Cycle](#) (page 167)

References

- Skiena, S. ‘Permutations.’ 1.1 in *Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica*. Reading, MA: Addison-Wesley, pp. 3-16, 1990.
- Knuth, D. E. *The Art of Computer Programming, Vol. 4: Combinatorial Algorithms*, 1st ed. Reading, MA: Addison-Wesley, 2011.
- 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
- D. L. Kreher, D. R. Stinson ‘*Combinatorial Algorithms*’ CRC Press, 1999
- Graham, R. L.; Knuth, D. E.; and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA: Addison-Wesley, 1994.
- <https://en.wikipedia.org/wiki/Permutation>
- https://en.wikipedia.org/wiki/Lehmer_code

property array_form

Return a copy of the attribute `_array_form` .. rubric:: Examples

```
>>> 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 $p[i] < p[i+1]$

Examples

```
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.ascents()
[1, 2]
```

See also:

[descents](#) (page 154), [inversions](#) (page 158), [min](#) (page 161), [max](#) (page 161)

atoms()

Returns all the elements of a permutation

Examples

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

property cardinality

Returns the number of all possible permutations.

Examples

```
>>> p = Permutation([0, 1, 2, 3])
>>> p.cardinality
24
```

See also:

[length](#) (page 160), [order](#) (page 162), [rank](#) (page 163), [size](#) (page 165)

commutator(x)

Return the commutator of self and x: $\sim x \sim \text{self} * x * \text{self}$

If f and g are part of a group, G, then the commutator of f and g is the group identity iff f and g commute, i.e. $fg == gf$.

Examples

```
>>> 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
... 
```

References

<https://en.wikipedia.org/wiki/Commutator>

commutes_with(*other*)

Checks if the elements are commuting.

Examples

```
>>> a = Permutation([1, 4, 3, 0, 2, 5])
>>> b = Permutation([0, 1, 2, 3, 4, 5])
>>> a.commutates_with(b)
True
>>> b = Permutation([2, 3, 5, 4, 1, 0])
>>> a.commutates_with(b)
False
```

property cycle_structure

Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

Examples

```
>>> Permutation.print_cyclic = True
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation(0, 4, 3)(1, 2)(5, 6).cycle_structure
{2: 2, 3: 1}
```

property cycles

Returns the number of cycles contained in the permutation (including singletons).

Examples

```
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation(0, 1)(2, 3).cycles
2
```

See also:

[*diofant.functions.combinatorial.numbers.stirling*](#) (page 316)

property `cyclic_form`

This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

Examples

```
>>> 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]]
```

See also:

[*array_form*](#) (page 151), [*full_cyclic_form*](#) (page 155)

method `descents()`

Returns the positions of descents in a permutation, ie, the location where $p[i] > p[i+1]$

Examples

```
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.descents()
[0, 3]
```

See also:

[*ascents*](#) (page 152), [*inversions*](#) (page 158), [*min*](#) (page 161), [*max*](#) (page 161)

classmethod `from_inversion_vector(inversion)`

Calculates the permutation from the inversion vector.

Examples

```
>>> 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

```
>>> 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)7(1, 3)
```

property full_cyclic_form

Return permutation in cyclic form including singletons.

Examples

```
>>> 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)

Examples

```
>>> 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
```

See also:

[get_precedence_matrix](#) (page 156), [get_precedence_distance](#) (page 156), [get_adjacency_matrix](#) (page 155)

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 .

Examples

```
>>> 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]])
```

See also:

[get_precedence_matrix](#) (page 156), [get_precedence_distance](#) (page 156),
[get_adjacency_distance](#) (page 155)

get_positional_distance(*other*)

Computes the positional distance between two permutations.

Examples

```
>>> 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
```

See also:

[get_precedence_distance](#) (page 156), [get_adjacency_distance](#) (page 155)

get_precedence_distance(*other*)

Computes the precedence distance between two permutations.

Suppose p and p' represent n jobs. The precedence metric counts the number of times a job j is preceded by job i in both p and p' . This metric is commutative.

Examples

```
>>> 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
```

See also:

[get_precedence_matrix](#) (page 156), [get_adjacency_matrix](#) (page 155),
[get_adjacency_distance](#) (page 155)

get_precedence_matrix()

Gets the precedence matrix. This is used for computing the distance between two permutations.

Examples

```
>>> p = Permutation.josephus(3, 6, 1)
>>> Permutation.print_cyclic = False
>>> 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, 1, 1, 0],
[1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 0]])
```

See also:

[get_precedence_distance](#) (page 156), [get_adjacency_matrix](#) (page 155), [get_adjacency_distance](#) (page 155)

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

```
>>> 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.

Examples

```
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the i -th element cycling through $0..i$.

```
>>> p = Permutation(2)
>>> Permutation.print_cyclic = False
>>> while p:
...     print(f'{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
```

See also:

from_inversion_vector (page 154)

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.

References

- <https://www.cp.eng.chula.ac.th/~prabhas/teaching/algo/algo2008/count-inv.htm>

Examples

```
>>> p = Permutation([0, 1, 2, 3, 4, 5])
>>> p.inversions()
0
>>> Permutation([3, 2, 1, 0]).inversions()
6
```

See also:

descents (page 154), *ascents* (page 152), *min* (page 161), *max* (page 161)

property is_Empty

Checks to see if the permutation is a set with zero elements

Examples

```
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

See also:

is_Singleton (page 159)

property is_Identity

Returns True if the Permutation is an identity permutation.

Examples

```

>>> 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

```

See also:

[order](#) (page 162)

property `is_Singleton`

Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

Examples

```

>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False

```

See also:

[is_Empty](#) (page 158)

property `is_even`

Checks if a permutation is even.

Examples

```

>>> p = Permutation([0, 1, 2, 3])
>>> p.is_even
True
>>> p = Permutation([3, 2, 1, 0])
>>> p.is_even
True

```

See also:

[is_odd](#) (page 159)

property `is_odd`

Checks if a permutation is odd.

Examples

```
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_odd
False
>>> p = Permutation([3, 2, 0, 1])
>>> p.is_odd
True
```

See also:

[*is_even*](#) (page 159)

`classmethod josephus(m, n, s=1)`

Return as a permutation the shuffling of `range(n)` using the Josephus scheme in which every `m`-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter `s` stops the selection process when there are `s` items remaining and these are selected by continuing the selection, counting by 1 rather than by `m`.

Consider selecting every 3rd item from 6 until only 2 remain:

<u>choices</u>	<u>chosen</u>
012345	
01345	2
0134	25
0134	253
014	2531
04	25314
0	253140

Examples

```
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]
```

References

- https://en.wikipedia.org/wiki/Flavius_Josephus
- https://en.wikipedia.org/wiki/Josephus_problem

`length()`

Returns the number of integers moved by a permutation.

Examples

```
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4
```

See also:

[*min*](#) (page 161), [*max*](#) (page 161), [*support*](#) (page 166), [*cardinality*](#) (page 152), [*order*](#) (page 162), [*rank*](#) (page 163), [*size*](#) (page 165)

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

```
>>> 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.

Examples

```
>>> p = Permutation([1, 0, 2, 3, 4])
>>> p.max()
1
```

See also:

[min](#) (page 161), [descents](#) (page 154), [ascents](#) (page 152), [inversions](#) (page 158)

min()

The minimum element moved by the permutation.

Examples

```
>>> p = Permutation([0, 1, 4, 3, 2])
>>> p.min()
2
```

See also:

[max](#) (page 161), [descents](#) (page 154), [ascents](#) (page 152), [inversions](#) (page 158)

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.

Examples

```
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0])
>>> p.rank()
17
>>> p = p.next_lex()
>>> p.rank()
18
```

See also:

[rank](#) (page 163), [unrank_lex](#) (page 166)

`next_nonlex()`

Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.

Examples

```
>>> 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
```

See also:

[rank_nonlex](#) (page 163), [unrank_nonlex](#) (page 166)

`next_trotterjohnson()`

Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4.

Examples

```
>>> 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
```

See also:

[rank_trotterjohnson](#) (page 164), [unrank_trotterjohnson](#) (page 167)

`order()`

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

Examples

```
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

See also:

[is_Identity](#) (page 158), [cardinality](#) (page 152), [length](#) (page 160), [rank](#) (page 163), [size](#) (page 165)

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 $p[x] < p[y]$.

Examples

```
>>> p = Permutation([0, 1, 2, 3])
>>> p.parity()
0
>>> p = Permutation([3, 2, 0, 1])
>>> p.parity()
1
```

See also:

[_af_parity](#) (page 168)

classmethod random(n)

Generates a random permutation of length n .

Uses the underlying Python pseudo-random number generator.

Examples

```
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0, 1]))
True
```

rank()

Returns the lexicographic rank of the permutation.

Examples

```
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

See also:

[next_lex](#) (page 161), [unrank_lex](#) (page 166), [cardinality](#) (page 152), [length](#) (page 160), [order](#) (page 162), [size](#) (page 165)

rank_nonlex(*inv_perm=None*)

This is a linear time ranking algorithm that does not enforce lexicographic order [3].

Examples

```
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_nonlex()
23
```

See also:

[*next_nonlex*](#) (page 162), [*unrank_nonlex*](#) (page 166)

rank_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

Examples

```
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0, 2, 1, 3])
>>> p.rank_trotterjohnson()
7
```

See also:

[*unrank_trotterjohnson*](#) (page 167), [*next_trotterjohnson*](#) (page 162)

static **rmul**(*args)

Return product of Permutations [a, b, c, ...] as the Permutation whose *i*th value is $a(b(c(i)))$.

a, b, c, ... can be Permutation objects or tuples.

Examples

```
>>> 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]
```


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`.

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

```
>>> p = Permutation([2, 5, 7, 3, 6, 0, 1, 4, 8])
>>> p.runs()
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1, 3, 2, 0])
>>> q.runs()
[[1, 3], [2], [0]]
```

`signature()`

Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

The signature is calculated as $(-1)^{\text{number of inversions}}$

Examples

```
>>> p = Permutation([0, 1, 2])
>>> p.inversions()
0
>>> p.signature()
1
>>> q = Permutation([0, 2, 1])
>>> q.inversions()
1
>>> q.signature()
-1
```

See also:

[*inversions*](#) (page 158)

property `size`

Returns the number of elements in the permutation.

Examples

```
>>> Permutation([[3, 2], [0, 1]]).size
4
```

See also:

cardinality (page 152), *length* (page 160), *order* (page 162), *rank* (page 163)

`support()`

Return the elements in permutation, P , for which $P[i] \neq i$.

Examples

```
>>> 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]

Examples

```
>>> 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
```

References

1. https://en.wikipedia.org/wiki/Transposition_%28mathematics%29#Properties

`classmethod unrank_lex(size, rank)`

Lexicographic permutation unranking.

Examples

```
>>> Permutation.print_cyclic = False
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
>>> a
Permutation([0, 2, 4, 1, 3])
```

See also:

rank (page 163), *next_lex* (page 161)

classmethod unrank_nonlex(*n*, *r*)

This is a linear time unranking algorithm that does not respect lexicographic order [3].

Examples

```
>>> Permutation.print_cyclic = False
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

See also:

[next_nonlex](#) (page 162), [rank_nonlex](#) (page 163)

classmethod unrank_trotterjohnson(*size*, *rank*)

Trotter Johnson permutation unranking. See [4] section 2.4.

Examples

```
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

See also:

[rank_trotterjohnson](#) (page 164), [next_trotterjohnson](#) (page 162)

class diofant.combinatorics.permutations.**Cycle**(*args)

Wrapper around dict which provides the functionality of a disjoint cycle.

A cycle shows the rule to use to move subsets of elements to obtain a permutation. The Cycle class is more flexible than Permutation in that 1) all elements need not be present in order to investigate how multiple cycles act in sequence and 2) it can contain singletons:

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)]
```

Notes

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

```
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

See also:

[Permutation](#) (page 147)

`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

```
>>> 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]
```

`diofant.combinatorics.permutations._af_parity(pi)`

Computes the parity of a permutation in array form.

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 $p[x] < p[y]$.

Examples

```
>>> _af_parity([0, 1, 2, 3])
0
>>> _af_parity([3, 2, 0, 1])
1
```

See also:

[Permutation](#) (page 147)

Generators

`generators.symmetric()`

Generates the symmetric group of order n , S_n .

Examples

```
>>> Permutation.print_cyclic = True
>>> list(symmetric(3))
[Permutation(2), Permutation(1, 2), Permutation(2)(0, 1),
 Permutation(0, 1, 2), Permutation(0, 2, 1), Permutation(0, 2)]
```

`generators.cyclic()`

Generates the cyclic group of order n , C_n .

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)]
```

See also:

[dihedral](#) (page 169)

`generators.alternating()`

Generates the alternating group of order n , A_n .

Examples

```
>>> Permutation.print_cyclic = True
>>> list(alternating(3))
[Permutation(2), Permutation(0, 1, 2), Permutation(0, 2, 1)]
```

`generators.dihedral()`

Generates the dihedral group of order $2n$, D_n .

The result is given as a subgroup of S_n , except for the special cases $n=1$ (the group S_2) and $n=2$ (the Klein 4-group) where that's not possible and embeddings in S_2 and S_4 respectively are given.

Examples

```
>>> Permutation.print_cyclic = True
>>> list(dihedral(3))
[Permutation(2), Permutation(0, 2), Permutation(0, 1, 2),
 Permutation(1, 2), Permutation(0, 2, 1), Permutation(2)(0, 1)]
```

See also:

[*cyclic*](#) (page 169)

4.3.3 Permutation Groups

class `diofant.combinatorics.perm_groups.PermutationGroup(*args, **kwargs)`

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.

Examples

```
>>> Permutation.print_cyclic = True
```

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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']
```

See also:

diofant.combinatorics.polyhedron.Polyhedron (page 196), *diofant.combinatorics.permutations.Permutation* (page 147)

References

- [1] Holt, D., Eick, B., O’Brien, E. “Handbook of Computational Group Theory”
- [2] Seress, A. “Permutation Group Algorithms”
- [3] https://en.wikipedia.org/wiki/Schreier_vector
- [4] https://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] https://en.wikipedia.org/wiki/Block_permutation_group_theory
- [7] https://web.archive.org/web/20170105021515/http://www.algorithmist.com:80/index.php/Union_Find
- [8] https://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups
- [9] https://en.wikipedia.org/wiki/Center_group_theory
- [10] https://en.wikipedia.org/wiki/Centralizer_and_normalizer
- [11] https://groupprops.subwiki.org/wiki/Derived_subgroup
- [12] https://en.wikipedia.org/wiki/Nilpotent_group
- [13] <https://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf>

`__contains__(i)`

Return True if *i* is contained in `PermutationGroup`.

Examples

```
>>> p = Permutation(1, 2, 3)
>>> Permutation(3) in PermutationGroup(p)
True
```

`__eq__(other)`

Return True if PermutationGroup generated by elements in the group are same i.e they represent the same PermutationGroup.

Examples

```
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G = PermutationGroup([p, p**2])
>>> H = PermutationGroup([p**2, p])
>>> G.generators == H.generators
False
>>> G == H
True
```

`__mul__(other)`

Return the direct product of two permutation groups as a permutation group.

This implementation realizes the direct product by shifting the index set for the generators of the second group: so if we have G acting on n_1 points and H acting on n_2 points, $G*H$ acts on $n_1 + n_2$ points.

Examples

```
>>> G = CyclicGroup(5)
>>> H = G*G
>>> H
PermutationGroup([
  Permutation(9)(0, 1, 2, 3, 4),
  Permutation(5, 6, 7, 8, 9)])
>>> H.order()
25
```

`static __new__(cls, *args, **kwargs)`

The default constructor. Accepts Cycle and Permutation forms. Removes duplicates unless `dups` keyword is False.

`__union_find_merge(first, second, ranks, parents, not_rep)`

Merges two classes in a union-find data structure.

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. The class merging process uses union by rank as an optimization. ([7])

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, parents, the list of class sizes, ranks, and the list of elements that are not representatives, `not_rep`, are changed due to class merging.

See also:

[`minimal_block`](#) (page 187), [`_union_find_rep`](#) (page 173)

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[7] https://web.archive.org/web/20170105021515/http://www.algorithmist.com:80/index.php/Union_Find

`_union_find_rep(num, parents)`

Find representative of a class in a union-find data structure.

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. After the representative of the class to which `num` belongs is found, path compression is performed as an optimization ([7]).

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, parents, is altered due to path compression.

See also:

[`minimal_block`](#) (page 187), [`_union_find_merge`](#) (page 172)

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[7] https://web.archive.org/web/20170105021515/http://www.algorithmist.com:80/index.php/Union_Find

property base

Return a base from the Schreier-Sims algorithm.

For a permutation group G , a base is a sequence of points $B = (b_1, b_2, \dots, b_k)$ such that no element of G apart from the identity fixes all the points in B . The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

An alternative way to think of B is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.

Examples

```
>>> G = PermutationGroup([Permutation(0, 1, 3)(2, 4)])
>>> G.base
[0, 2]
```

See also:

[strong_gens](#) (page 194), [basic_transversals](#) (page 175), [basic_orbits](#) (page 175), [basic_stabilizers](#) (page 175)

baseswap(*base*, *strong_gens*, *pos*, *randomized=False*, *transversals=None*, *basic_orbits=None*, *strong_gens_distr=None*)

Swap two consecutive base points in base and strong generating set.

If a base for a group G is given by (b_1, b_2, \dots, b_k) , this function returns a base $(b_1, b_2, \dots, b_{\{i+1\}}, b_i, \dots, b_k)$, where i is given by *pos*, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.

Parameters

- **base**, **strong_gens** – The base and strong generating set.
- **pos** – The position at which swapping is performed.
- **randomized** – A switch between randomized and deterministic version.
- **transversals** – The transversals for the basic orbits, if known.
- **basic_orbits** – The basic orbits, if known.
- **strong_gens_distr** – The strong generators distributed by basic stabilizers, if known.

Returns

(*base*, *strong_gens*) – *base* is the new base, and *strong_gens* is a generating set relative to it.

Examples

```
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> 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
```

See also:

[schreier_sims](#) (page 191)

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, $|\beta_{i+1}^{\langle T \rangle}|$ should be replaced by $|\beta_i^{\langle T \rangle}|$, and the same for the discussion of the algorithm.

property `basic_orbits`

Return the basic orbits relative to a base and strong generating set.

If (b_1, b_2, \dots, b_k) is a base for a group G , and $G^{\{i\}} = G_{\{b_1, b_2, \dots, b_{i-1}\}}$ is the i -th basic stabilizer (so that $G^{\{1\}} = G$), the i -th basic orbit relative to this base is the orbit of b_i under $G^{\{i\}}$. See [1], pp. 87-89 for more information.

Examples

```
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[0, 1, 2, 3], [1, 2, 3], [2, 3]]
```

See also:

[base](#) (page 173), [strong_gens](#) (page 194), [basic_transversals](#) (page 175), [basic_stabilizers](#) (page 175)

property `basic_stabilizers`

Return a chain of stabilizers relative to a base and strong generating set.

The i -th basic stabilizer $G^{\{i\}}$ relative to a base (b_1, b_2, \dots, b_k) is $G_{\{b_1, b_2, \dots, b_{i-1}\}}$. For more information, see [1], pp. 87-89.

Examples

```
>>> 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)])
```

See also:

[base](#) (page 173), [strong_gens](#) (page 194), [basic_orbits](#) (page 175), [basic_transversals](#) (page 175)

property `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.

Examples

```
>>> 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)}]
```

See also:

[strong_gens](#) (page 194), [base](#) (page 173), [basic_orbits](#) (page 175), [basic_stabilizers](#) (page 175)

center()

Return the center of a permutation group.

The center for a group G is defined as $Z(G) = \{z \in G \mid \text{forall } g \in G, zg = gz\}$, 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]).

Examples

```
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

See also:

[centralizer](#) (page 176)

Notes

This is a naive implementation that is a straightforward application of `centralizer()`.

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 \mid gs = sg \text{ 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

Examples

```
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

See also:

[subgroup_search](#) (page 194)

Notes

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group G .

`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).

Examples

```
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

See also:

[derived_subgroup](#) (page 180)

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)

`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.

Examples

```
>>> Permutation.print_cyclic = True
```

```
>>> 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
```

See also:

[*coset_factor*](#) (page 178), [*diofant.core.basic.Basic.has*](#) (page 48)

coset_factor(*g*, *factor_index*=False)

Return *G*'s (self's) coset factorization of *g*

If *g* is an element of *G* then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in *f* are those for which the product gives *g*: $g = f[n] * \dots * f[1] * f[0]$ where $n = \text{len}(B)$ and $B = G.\text{base}$. *f*[*i*] is one of the permutations in *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

```
>>> 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 u_1 and u_2 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`.

Examples

```
>>> 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])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
Permutation(7)(2, 4)(3, 5)
```

See also:

[`coset_factor`](#) (page 178)

`coset_unrank(rank, af=False)`

Unrank using Schreier-Sims representation.

`coset_unrank` is the inverse operation of `coset_rank` if $0 \leq \text{rank} < \text{order}$; otherwise it returns `None`.

`property 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()`.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)]
```

See also:

[order](#) (page 190)

`derived_series()`

Return the derived series for the group.

The derived series for a group G is defined as $G = G_0 > G_1 > G_2 > \dots$ where $G_i = [G_{i-1}, G_{i-1}]$, i.e. G_i is the derived subgroup of G_{i-1} , for $i \in \mathbb{N}$. When we have $G_k = G_{k-1}$ for some $k \in \mathbb{N}$, the series terminates.

Returns

- A list of permutation groups containing the members of the derived
- series in the order $G = G_0, G_1, G_2, \dots$.

Examples

```
>>> 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
```

See also:

[derived_subgroup](#) (page 180)

`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]).

Examples

```
>>> Permutation.print_cyclic = True
>>> 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]]
```

See also:

[derived_series](#) (page 180)

property elements

Returns all the elements of the permutation group in a list

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

```
>>> Permutation.print_cyclic = True
```

The permutation group given in the tetrahedron object is also true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group
True
```

Also the group generated by the permutations in the tetrahedron pgroup – even the first two – is a proper group:

```
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate()))
>>> J
PermutationGroup([
  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)])
>>> J.is_group
True
```

generate_dimino(*af*=False)

Yield group elements using Dimino's algorithm

If *af* == True it yields the array form of the permutations

References

[1] The Implementation of Various Algorithms for Permutation Groups in the Computer Algebra System: AXIOM, N.J. Doye, M.Sc. Thesis

Examples

```
>>> Permutation.print_cyclic = True
>>> 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

```
>>> Permutation.print_cyclic = True
>>> 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]]
```

property generators

Returns the generators of the group.

Examples

```
>>> Permutation.print_cyclic = True
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.generators
[Permutation(1, 2), Permutation(2)(0, 1)]
```

property is_abelian

Test if the group is Abelian.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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 ϵ .

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

```
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

See also:

[`diofant.combinatorics.util._check_cycles_alt_sym`](#) (page 216)

property 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]).

Examples

```
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
>>> S.is_nilpotent
False
```

See also:

[`lower_central_series`](#) (page 186), [`is_solvable`](#) (page 184)

is_normal(*gr*, *strict*=True)

Test if $G=\text{self}$ is a normal subgroup of gr .

G is normal in gr if for each g_2 in G , g_1 in gr , $g = g_1 * g_2 * g_1^{-1}$ belongs to G . It is sufficient to check this for each g_1 in $gr.generator$ and g_2 in $G.generator$.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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).

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

```
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False
```

See also:

[`minimal_block`](#) (page 187), [`random_stab`](#) (page 191)

property **is_solvable**

Test if the group is solvable.

G is solvable if its derived series terminates with the trivial group ([1], p.29).

Examples

```
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True
```

See also:

[`is_nilpotent`](#) (page 183), [`derived_series`](#) (page 180)

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

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

```
>>> 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
```

property is_trivial

Test if the group is the trivial group.

This is true if the group contains only the identity permutation.

Examples

```
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True
```

lower_central_series()

Return the lower central series for the group.

The lower central series for a group G is the series $G = G_0 > G_1 > G_2 > \dots$ where $G_k = [G, G_{k-1}]$, i.e. every term after the first is equal to the commutator of G and the previous term in G ([1], p.29).

Returns

- A list of permutation groups in the order
- $G = G_0, G_1, G_2, \dots$

Examples

```
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

See also:

[commutator](#) (page 177), [derived_series](#) (page 180)

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.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)
```

See also:[random](#) (page 191)**property `max_div`**

Maximum proper divisor of the degree of a permutation group.

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 `diofant.ntheory` is used. This function is also used as an optimization tool for the functions `minimal_block` and `_union_find_merge`.

Examples

```
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2
```

See also:[minimal_block](#) (page 187), [_union_find_merge](#) (page 172)**`minimal_block(points)`**

For a transitive group, finds the block system generated by points.

If a group G acts on a set S , a nonempty subset B of S is called a block under the action of G if for all g in G we have $gB = B$ (g fixes B) or gB and B have no common points (g moves B entirely). ([1], p.23; [6]).

The distinct translates gB of a block B for g in G partition the set S and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides $|S|$ ([1], p.23). A G -congruence is an equivalence relation \sim on the set S such that $a \sim b$ implies $g(a) \sim g(b)$ for all g in G . For a transitive group, the equivalence classes of a G -congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the G -congruence generated by the pairs (p_0, p_1) , (p_0, p_2) , \dots , (p_0, p_{k-1}) which is the same as finding the maximal block system (i.e., the one with minimum block size) such that p_0, \dots, p_{k-1} are in the same block ([1], p.83).

It is an implementation of Atkinson's algorithm, as suggested in [1], and manipulates an equivalence relation on the set S using a union-find data structure. The running time is just above $O(|points| |S|)$. ([1], pp. 83-87; [7]).

Examples

```
>>> 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]
```

See also:

[_union_find_rep](#) (page 173), [_union_find_merge](#) (page 172), [is_transitive](#) (page 185), [is_primitive](#) (page 184)

normal_closure(other, k=10)

Return the normal closure of a subgroup/set of permutations.

If S is a subset of a group G , the normal closure of A in G is defined as the intersection of all normal subgroups of G that contain A ([1], p.14). Alternatively, it is the group generated by the conjugates $x^{-1}yx$ for x a generator of G and y a generator of the subgroup $\langle S \rangle$ generated by S (for some chosen generating set for $\langle S \rangle$) ([1], p.73).

Parameters

- **other** – a subgroup/list of permutations/single permutation
- **k** – an implementation-specific parameter that determines the number of conjugates that are adjoined to other at once

Examples

```
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```

See also:

[commutator](#) (page 177), [derived_subgroup](#) (page 180), [random_pr](#) (page 191)

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.

orbit(alpha, action='tuples')

Compute the orbit of $\alpha \setminus \{g(\alpha) \mid g \in G\}$ 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 α can be a single point, or a list of points.

If α is a single point, the ordinary orbit is computed. if α 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

Examples

```
>>> a = Permutation([1, 2, 0, 4, 5, 6, 3])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
{0, 1, 2}
>>> G.orbit([0, 4], 'union')
{0, 1, 2, 3, 4, 5, 6}
```

See also:

[orbit_transversal](#) (page 189)

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

Examples

```
>>> Permutation.print_cyclic = True
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
Permutation(0, 4, 1, 2, 3)
```

See also:

[schreier_vector](#) (page 193)

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 $Orb = \{g(\alpha) \mid g \in G\}$ is a set $\{g_\beta(\alpha) \mid g_\beta(\alpha) = \beta\}$ for $\beta \in Orb$. Note that there may be more than one possible transversal. If *pairs* is set to True, it returns the list of pairs (β, g_β) . For a proof of correctness, see [1], p.79

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)]
```

See also:

[orbit](#) (page 188)

orbits(*rep=False*)

Return the orbits of self, ordered according to lowest element in each orbit.

Examples

```
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[{0, 2, 3, 4, 6}, {1, 5}]
```

order()

Return the order of the group: the number of permutations that can be generated from elements of the group.

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

Examples

```
>>> 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
```

See also:

[degree](#) (page 179)

pointwise_stabilizer(*points, incremental=True*)

Return the pointwise stabilizer for a set of points.

For a permutation group G and a set of points $\{p_1, p_2, \dots, p_k\}$, the pointwise stabilizer of p_1, p_2, \dots, p_k is defined as $G_{\{p_1, \dots, p_k\}} = \{g \in G \mid g(p_i) = p_i \text{ for all } i \in \{1, 2, \dots, k\}\}$ ([1], p20). It is a subgroup of G .

Examples

```
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True
```

See also:

[stabilizer](#) (page 193), [schreier_sims_incremental](#) (page 191)

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.

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`.

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 191), [orbit_rep](#) (page 189)

schreier_sims()

Schreier-Sims algorithm.

It computes the generators of the chain of stabilizers $G > G_{\{b_1\}} > \dots > G_{\{b_1, \dots, b_r\}} > 1$ in which $G_{\{b_1, \dots, b_i\}}$ stabilizes b_1, \dots, b_i , and the corresponding s cosets. An element of the group can be written as the product $h_1 * \dots * h_s$.

We use the incremental Schreier-Sims algorithm.

Examples

```
>>> 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.

Examples

```
>>> from diofant.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]
```

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.

See also:

[*schreier_sims*](#) (page 191), [*schreier_sims_random*](#) (page 192)

`schreier_sims_random`(*base=None, gens=None, consec_succ=10, _random_prec=None*)

Randomized Schreier-Sims algorithm.

The randomized Schreier-Sims algorithm takes the sequence *base* and the generating set *gens*, and extends *base* to a base, and *gens* to a strong generating set relative to that base with probability of a wrong answer at most $2^{-\text{consec_succ}}$, provided the random generators are sufficiently random.

Parameters

- **base** - The sequence to be extended to a base.
- **gens** - The generating set to be extended to a strong generating set.
- **consec_succ** - The parameter defining the probability of a wrong answer.
- **_random_prec** - An internal parameter used for testing purposes.

Returns

(*base*, *strong_gens*) - *base* is the base and *strong_gens* is the strong generating set relative to it.

Examples

```
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

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 $1 - 1/\text{consec_succ}$.

See also:

[`schreier_sims`](#) (page 191)

`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.

Examples

```
>>> 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]
```

See also:

[`orbit`](#) (page 188)

stabilizer(alpha)

Return the stabilizer subgroup of α .

The stabilizer of α is the group $G_\alpha = \{g \in G \mid g(\alpha) = \alpha\}$. For a proof of correctness, see [1], p.79.

Examples

```
>>> Permutation.print_cyclic = True
>>> G = DihedralGroup(6)
>>> G.stabilizer(5)
PermutationGroup([
  Permutation(5)(0, 4)(1, 3),
  Permutation(5)])
```

See also:

[orbit](#) (page 188)

property strong_gens

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

A generating set $S = \{g_1, g_2, \dots, g_t\}$ for a permutation group G is a strong generating set relative to the sequence of points (referred to as a “base”) (b_1, b_2, \dots, b_k) if, for $1 \leq i \leq k$ we have that the intersection of the pointwise stabilizer $G^{(i+1)} := G_{\{b_1, b_2, \dots, b_i\}}$ with S generates the pointwise stabilizer $G^{(i+1)}$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

Examples

```
>>> D = DihedralGroup(4)
>>> D.strong_gens
[Permutation(0, 1, 2, 3), Permutation(0, 3)(1, 2), Permutation(1, 3)]
>>> D.base
[0, 1]
```

See also:

[base](#) (page 173), [basic_transversals](#) (page 175), [basic_orbits](#) (page 175), [basic_stabilizers](#) (page 175)

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`.

Examples

```
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(7)
>>> def prop_even(x):
...     return 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
```

Notes

This function is extremely lengthy and complicated and will require some careful attention. The implementation is described in [1], pp. 114-117, and the comments for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits all members of the supergroup. However, there are a lot of tests which are used to prune the search tree, and users can define their own tests via the `tests` parameter, so in practice, and for some computations, it's not terrible.

A crucial part in the procedure is the frequent base change performed (this is line 11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentions that this can be done by using `.baseswap(...)`, however the current implementation uses a more straightforward way to find the next basic stabilizer - calling the function `.stabilizer(...)` on the previous basic stabilizer.

property transitivity_degree

Compute the degree of transitivity of the group.

A permutation group G acting on $\Omega = \{0, 1, \dots, n-1\}$ is k -fold transitive, if, for any k points $(a_1, a_2, \dots, a_k) \in \Omega$ and any k points $(b_1, b_2, \dots, b_k) \in \Omega$ there exists $g \in G$ such that $g(a_1)=b_1, g(a_2)=b_2, \dots, g(a_k)=b_k$. The degree of transitivity of G is the maximum k such that G is k -fold transitive. ([8])

Examples

```
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```

See also:

is_transitive (page 185), *orbit* (page 188)

4.3.4 Polyhedron

class diofant.combinatorics.polyhedron.**Polyhedron**(*corners*, *faces*=[], *pgroup*=[])

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

- <https://mathworld.wolfram.com/PolyhedralGroup.html>

property array_form

Return the indices of the corners.

The indices are given relative to the original position of corners.

Examples

```
>>> 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]
>>> tetrahedron.reset()
```

See also:

corners (page 196), *cyclic_form* (page 197)

property corners

Get the corners of the Polyhedron.

The method vertices is an alias for corners.

Examples

```
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

See also:

[array_form](#) (page 196), [cyclic_form](#) (page 197)

property `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 196), [array_form](#) (page 196)

property `edges`

Given the faces of the polyhedra we can get the edges.

Examples

```
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
{(0, 1), (0, 2), (1, 2)}
```

property `faces`

Get the faces of the Polyhedron.

property `pgroup`

Get the permutations of the Polyhedron.

`reset()`

Return corners to their original positions.

Examples

```
>>> tetrahedron.corners
(0, 1, 2, 3)
>>> tetrahedron.rotate(0)
>>> tetrahedron.corners
(0, 2, 3, 1)
>>> tetrahedron.reset()
>>> tetrahedron.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

```
>>> 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
```

property size

Get the number of corners of the `Polyhedron`.

property vertices

Get the corners of the `Polyhedron`.

The method `vertices` is an alias for `corners`.

Examples

```
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

See also:

[*array_form*](#) (page 196), [*cyclic_form*](#) (page 197)

4.3.5 Prufer Sequences

class diofant.combinatorics.prufer.**Prufer**(*args, **kw_args)

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

- <https://mathworld.wolfram.com/LabeledTree.html>

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

```
>>> 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.

Examples

```
>>> 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
```

See also:

[prufer_rank](#) (page 200), [rank](#) (page 200), [prev](#) (page 200), [size](#) (page 201)

property nodes

Returns the number of nodes in the tree.

Examples

```
>>> 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.

Examples

```
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer((1, 2, 0))
>>> b.rank
35
```

See also:

[prufer_rank](#) (page 200), [rank](#) (page 200), [next](#) (page 199), [size](#) (page 201)

prufer_rank()

Computes the rank of a Prufer sequence.

Examples

```
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

See also:

[rank](#) (page 200), [next](#) (page 199), [prev](#) (page 200), [size](#) (page 201)

property prufer_repr

Returns Prufer sequence for the Prufer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prufer sequence is the list of recorded nodes.

Examples

```
>>> 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]
```

See also:

[to_prufer](#) (page 201)

property rank

Returns the rank of the Prufer sequence.

Examples

```
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

See also:

[prufer_rank](#) (page 200), [next](#) (page 199), [prev](#) (page 200), [size](#) (page 201)

property size

Return the number of possible trees of this Prufer object.

Examples

```
>>> Prufer([0]*4).size == Prufer([6]*4).size == 1296
True
```

See also:

[prufer_rank](#) (page 200), [rank](#) (page 200), [next](#) (page 199), [prev](#) (page 200)

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.

Examples

```
>>> 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]
```

See also:

[prufer_repr](#) (page 200)

returns Prufer sequence of a Prufer object.

static to_tree(prufer)

Return the tree (as a list of edges) of the given Prufer sequence.

Examples

```
>>> 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]]
```

References

- <https://hamberg.no/erlend/posts/2010-11-06-prufer-sequence-compact-tree-representation.html>

See also:

`tree_repr` (page 202)

returns tree representation of a Prufer object.

property `tree_repr`

Returns the tree representation of the Prufer object.

Examples

```
>>> 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]]
```

See also:

`to_tree` (page 201)

classmethod `unrank(rank, n)`

Finds the unranked Prufer sequence.

Examples

```
>>> Prufer.unrank(0, 4)
Prufer((0, 0))
```

4.3.6 Subsets

class `diofant.combinatorics.subsets.Subset(subset, superset)`

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

```
>>> 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.

Examples

```
>>> Subset.bitlist_from_subset(['c', 'd'], ['a', 'b', 'c', 'd'])
'0011'
```

See also:

[subset_from_bitlist](#) (page 206)

property cardinality

Returns the number of all possible subsets.

Examples

```
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.cardinality
16
```

See also:

[subset](#) (page 206), [superset](#) (page 207), [size](#) (page 206), [superset_size](#) (page 207)

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.

Examples

```
>>> 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
[]
```

See also:

[next_binary](#) (page 203), [prev_binary](#) (page 204)

iterate_graycode(k)

Helper function used for [prev_gray](#) and [next_gray](#). It performs k step overs to get the respective Gray codes.

Examples

```
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

See also:

[next_gray](#) (page 204), [prev_gray](#) (page 204)

next_binary()

Generates the next binary ordered subset.

Examples

```
>>> 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
[]
```

See also:

[*prev_binary*](#) (page 204), [*iterate_binary*](#) (page 203)

next_gray()

Generates the next Gray code ordered subset.

Examples

```
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.next_gray().subset
[1, 3]
```

See also:

[*iterate_graycode*](#) (page 203), [*prev_gray*](#) (page 204)

next_lexicographic()

Generates the next lexicographically ordered subset.

Examples

```
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
[]
```

See also:

[*prev_lexicographic*](#) (page 205)

prev_binary()

Generates the previous binary ordered subset.

Examples

```
>>> 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']
```

See also:

[*next_binary*](#) (page 203), [*iterate_binary*](#) (page 203)

prev_gray()

Generates the previous Gray code ordered subset.

Examples

```
>>> a = Subset([2, 3, 4], [1, 2, 3, 4, 5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

See also:

[*iterate_graycode*](#) (page 203), [*next_gray*](#) (page 204)

prev_lexicographic()

Generates the previous lexicographically ordered subset.

Examples

```
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['d']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['c']
```

See also:

[*next_lexicographic*](#) (page 204)

property rank_binary

Computes the binary ordered rank.

Examples

```
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

See also:

[*iterate_binary*](#) (page 203), [*unrank_binary*](#) (page 207)

property rank_gray

Computes the Gray code ranking of the subset.

Examples

```
>>> 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
```

See also:

[*iterate_graycode*](#) (page 203), [*unrank_gray*](#) (page 207)

property rank_lexicographic

Computes the lexicographic ranking of the subset.

Examples

```
>>> 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
```

property size

Gets the size of the subset.

Examples

```
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.size
2
```

See also:

[subset](#) (page 206), [superset](#) (page 207), [superset_size](#) (page 207), [cardinality](#) (page 203)

property subset

Gets the subset represented by the current instance.

Examples

```
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.subset
['c', 'd']
```

See also:

[superset](#) (page 207), [size](#) (page 206), [superset_size](#) (page 207), [cardinality](#) (page 203)

classmethod subset_from_bitlist(*super_set*, *bitlist*)

Gets the subset defined by the bitlist.

Examples

```
>>> Subset.subset_from_bitlist(['a', 'b', 'c', 'd'], '0011').subset
['c', 'd']
```

See also:

[bitlist_from_subset](#) (page 202)

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

```
>>> 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)
[]
```

property `superset`

Gets the superset of the subset.

Examples

```
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset
['a', 'b', 'c', 'd']
```

See also:

[*subset*](#) (page 206), [*size*](#) (page 206), [*superset_size*](#) (page 207), [*cardinality*](#) (page 203)

property `superset_size`

Returns the size of the superset.

Examples

```
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset_size
4
```

See also:

[*subset*](#) (page 206), [*superset*](#) (page 207), [*size*](#) (page 206), [*cardinality*](#) (page 203)

classmethod `unrank_binary(rank, superset)`

Gets the binary ordered subset of the specified rank.

Examples

```
>>> Subset.unrank_binary(4, ['a', 'b', 'c', 'd']).subset
['b']
```

See also:

[*iterate_binary*](#) (page 203), [*rank_binary*](#) (page 205)

classmethod `unrank_gray(rank, superset)`

Gets the Gray code ordered subset of the specified rank.

Examples

```
>>> Subset.unrank_gray(4, ['a', 'b', 'c']).subset  
['a', 'b']  
>>> Subset.unrank_gray(0, ['a', 'b', 'c']).subset  
[]
```

See also:

[iterate_graycode](#) (page 203), [rank_gray](#) (page 205)

`subsets.ksubsets(k)`

Finds the subsets of size k in lexicographic order.

This uses the `itertools` generator.

Examples

```
>>> 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)]
```

See also:

[Subset](#) (page 202)

4.3.7 Gray Code

class `diofant.combinatorics.graycode.GrayCode(n, *args, **kw_args)`

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

- Nijenhuis, A. and Wilf, H.S. (1978). Combinatorial Algorithms. Academic Press.
- Knuth, D. (2011). The Art of Computer Programming, Vol 4 Addison Wesley

Examples

```
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100',
 '1100', '1101', '1111', '1110', '1010', '1011', '1001', '1000']
```

property `current`

Returns the currently referenced Gray code as a bit string.

Examples

```
>>> 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

Examples

```
>>> 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']
```

See also:

[skip](#) (page 210)

property `n`

Returns the dimension of the Gray code.

Examples

```
>>> 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

```
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

property 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

- <http://statweb.stanford.edu/~susan/courses/s208/node12.html>

Examples

```
>>> 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'
```

See also:

[unrank](#) (page 211)

property selections

Returns the number of bit vectors in the Gray code.

Examples

```
>>> a = GrayCode(3)
>>> a.selections
8
```

skip()

Skips the bit generation.

Examples

```
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...     if i == '010':
...         a.skip()
...         print(i)
000
001
011
010
111
101
100
```

See also:

[generate_gray](#) (page 209)

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.

Examples

```
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

See also:

[rank](#) (page 210)

`graycode.random_bitstring()`

Generates a random bitlist of length n.

Examples

```
>>> random_bitstring(3)
'110'
```

`graycode.gray_to_bin()`

Convert from Gray coding to binary coding.

We assume big endian encoding.

Examples

```
>>> gray_to_bin('100')
'111'
```

See also:

[bin_to_gray](#) (page 211)

Examples

```
>>> G = SymmetricGroup(4)
>>> G.is_group
True
>>> G.order()
24
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],
 [1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],
 [2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],
 [3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
 [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1]]
```

See also:

[CyclicGroup](#) (page 213), [DihedralGroup](#) (page 213), [AlternatingGroup](#) (page 214)

References

[1] https://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations

`diofant.combinatorics.named_groups.CyclicGroup(n)`

Generates the cyclic group of order n as a permutation group.

The generator taken is the n -cycle $(0\ 1\ 2\ \dots\ n-1)$ (in cycle notation). After the group is generated, some of its basic properties are set.

Examples

```
>>> G = CyclicGroup(6)
>>> G.is_group
True
>>> 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]]
```

See also:

[SymmetricGroup](#) (page 212), [DihedralGroup](#) (page 213), [AlternatingGroup](#) (page 214)

`diofant.combinatorics.named_groups.DihedralGroup(n)`

Generates the dihedral group D_n as a permutation group.

The dihedral group D_n is the group of symmetries of the regular n -gon. The generators taken are the n -cycle $a = (0\ 1\ 2\ \dots\ n-1)$ (a rotation of the n -gon) and $b = (0\ n-1)(1\ n-2)\dots$ (a reflection of the n -gon) in cycle notation. It is easy to see that these satisfy $a^n = b^2 = 1$ and $bab = a^{-1}$ so they indeed generate D_n (See [1]). After the group is generated, some of its basic properties are set.

Examples

```
>>> G = DihedralGroup(5)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> [perm.cyclic_form for perm in a]
[[[]], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]], [[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]], [[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]], [[0, 3], [1, 2]]]
```

See also:

[SymmetricGroup](#) (page 212), [CyclicGroup](#) (page 213), [AlternatingGroup](#) (page 214)

References

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

`diofant.combinatorics.named_groups.AlternatingGroup(n)`

Generates the alternating group on *n* elements as a permutation group.

For $n > 2$, the generators taken are $(0\ 1\ 2)$, $(0\ 1\ 2\ \dots\ n-1)$ for *n* odd and $(0\ 1\ 2)$, $(1\ 2\ \dots\ n-1)$ for *n* even (See [1], p.31, ex.6.9.). After the group is generated, some of its basic properties are set. The cases $n = 1, 2$ are handled separately.

Examples

```
>>> G = AlternatingGroup(4)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> len(a)
12
>>> all(perm.is_even for perm in a)
True
```

See also:

[SymmetricGroup](#) (page 212), [CyclicGroup](#) (page 213), [DihedralGroup](#) (page 213)

References

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

`diofant.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.

Examples

```
>>> Permutation.print_cyclic = True
>>> AbelianGroup(3, 4)
PermutationGroup([
    Permutation(6)(0, 1, 2),
    Permutation(3, 4, 5, 6)])
>>> _.is_group
True
```

See also:

diofant.combinatorics.group_constructs.DirectProduct (page 221)

References

- https://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups

4.3.9 Utilities

`diofant.combinatorics.util._base_ordering(base, degree)`

Order $\{0, 1, \dots, n-1\}$ so that base points come first and in order.

Parameters

- `base` - the base
- `degree` - the degree of the associated permutation group

Returns

- A list `base_ordering` such that `base_ordering[point]` is the
- number of point in the ordering.

Examples

```
>>> 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 \ll on the underlying set for a permutation group of degree n , $\{0, 1, \dots, n-1\}$, so that if (b_1, b_2, \dots, b_k) is a base we have $b_i \ll b_j$ whenever $i < j$ and $b_i \ll a$ for all $i \in \{1, 2, \dots, k\}$ and a is not in the base. The idea is developed and applied to backtracking algorithms in [1], pp.108-132. The points that are not in the base are taken in increasing order.

References

- Holt, D., Eick, B., O’Brien, E. “Handbook of computational group theory”

`diofant.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 `diofant.combinatorics.perm_groups`.

Examples

```
>>> 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
```

See also:

[`diofant.combinatorics.perm_groups.PermutationGroup.is_alt_sym`](#) (page 182)

`diofant.combinatorics.util._distribute_gens_by_base(base, gens)`

Distribute the group elements `gens` by membership in basic stabilizers.

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

Parameters

- `base` - a sequence of points in $\{0, 1, \dots, n-1\}$
- `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*.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)]]
```

See also:

[_strong_gens_from_distr](#) (page 220), [_orbits_transversals_from_bsgs](#) (page 217), [_handle_precomputed_bsgs](#) (page 217)

```
diofant.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.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)]]])
```

See also:

[_orbits_transversals_from_bsgs](#) (page 217), [_distribute_gens_by_base](#) (page 216)

```
diofant.combinatorics.util._orbits_transversals_from_bsgs(base,
                                                            strong_gens_distr,
                                                            transver-
                                                            sals_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 switching between returning only the transversals/ both orbits and transversals

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)}])
```

See also:

[`_distribute_gens_by_base`](#) (page 216), [`_handle_precomputed_bsgs`](#) (page 217)

`diofant.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`.

Examples

```
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(15)
>>> base, strong_gens = S.schreier_sims_incremental()
>>> new_gens = _remove_gens(base, strong_gens)
>>> len(new_gens)
14
>>> _verify_bsgs(S, base, new_gens)
True
```

Notes

This procedure is outlined in [1],p.95.

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

`diofant.combinatorics.util._strip(g, base, orbits, transversals)`

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

This is done by treating the sequence *base* as an actual base, and the orbits *orbits* and transversals *transversals* as basic orbits and transversals relative to it.

This process is called "sifting". A sift is unsuccessful when a certain orbit element is not found or when after the sift the decomposition doesn't end with the identity element.

The argument *transversals* is a list of dictionaries that provides transversal elements for the orbits *orbits*.

Parameters

- `g` - permutation to be decomposed
- `base` - sequence of points
- `orbits` - a list in which the `i`-th entry is an orbit of `base[i]`
- under some subgroup of the pointwise stabilizer of
- `base[0], base[1], ..., base[i - 1]`. The groups themselves are implicit
- in this function since the only information we need is encoded in the orbits
- and transversals
- `transversals` - a list of orbit transversals associated with the orbits
- `orbits`.

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)
```

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"

See also:

[*diofant.combinatorics.perm_groups.PermutationGroup.schreier_sims*](#) (page 191),
[*diofant.combinatorics.perm_groups.PermutationGroup.schreier_sims_random*](#)
(page 192)

`diofant.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`

Examples

```
>>> Permutation.print_cyclic = True
>>> 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)]
```

See also:

[*_distribute_gens_by_base*](#) (page 216)

4.3.10 Group constructors

`diofant.combinatorics.group_constructs.DirectProduct(*groups)`

Returns the direct product of several groups as a permutation group.

This is implemented much like the `__mul__()` (page 172) 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).

Examples

```
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

See also:

`diofant.combinatorics.perm_groups.PermutationGroup.__mul__` (page 172)

4.3.11 Test Utilities

`diofant.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

```
>>> 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
```

`diofant.combinatorics.testutil._naive_list_centralizer(self, other, af=False)`

Return a list of elements for the centralizer of a subgroup/set/element.

This is a brute force implementation that goes over all elements of the group and checks for membership in the centralizer. It is used to test `.centralizer()` from `diofant.combinatorics.perm_groups`.

Examples

```
>>> D = DihedralGroup(4)
>>> _naive_list_centralizer(D, D)
[Permutation(3), Permutation(0, 2)(1, 3)]
```

See also:

[*diofant.combinatorics.perm_groups.PermutationGroup.centralizer*](#) (page 176)

`diofant.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.

Examples

```
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

See also:

[*diofant.combinatorics.perm_groups.PermutationGroup.schreier_sims*](#) (page 191)

`diofant.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 `diofant.combinatorics.perm_groups`

Examples

```
... AlternatingGroup) >>> S = SymmetricGroup(5) >>> A = AlternatingGroup(5) >>>
centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])]) >>> _verify_centralizer(S, A,
centr) True
```

See also:

[*_naive_list_centralizer*](#) (page 221), [*diofant.combinatorics.perm_groups.PermutationGroup.centralizer*](#) (page 176), [*_cmp_perm_lists*](#) (page 221)

`diofant.combinatorics.testutil._verify_normal_closure(group, arg, closure=None)`

Verify the normal closure of a subgroup/subset/element in a group.

This is used to test `diofant.combinatorics.perm_groups.PermutationGroup.normal_closure`

Examples

```
>>> S = SymmetricGroup(3)
>>> A = AlternatingGroup(3)
>>> _verify_normal_closure(S, A, closure=A)
True
```

See also:

[diofant.combinatorics.perm_groups.PermutationGroup.normal_closure](#)
(page 188)

4.3.12 Tensor Canonicalization

`diofant.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 [*d*₀, -*d*₀, *d*₁, -*d*₁, ...]
- ***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.

Notes

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.

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

$$g = [1, 3, 0, 5, 4, 2, 6, 7]$$

$$T_c = 0$$

```
>>> 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

$$T = f^c_{da} * f^f_{eb} * A^d_m * A^{mb} * A^a_n * A^{ne}$$

$$ord = [c, f, a, -a, b, -b, d, -d, e, -e, m, -m, n, -n]$$

$$g = [0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15]$$

The canonical tensor is $T_c = -f^{cab} * f^{fde} * A^m_a * A_{md} * A^n_b * A_{ne}$

$$can = [0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]$$

```
>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base_l, gens_l = get_symmetric_group_sgs(1)
>>> base_a, gens_a = bsgs_direct_product(base_l, gens_l, base_l, gens_l)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_a, gens_a, 4, 0)
```

(continues on next page)

(continued from previous page)

```
>>> 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]
```

`diofant.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

$$t = T(ind[g[0], \dots, ind[g[n-1]]],$$

where $n = \text{len}(\text{ind})$; g has size $n + 2$, the last two indices for the sign of the tensor (trick introduced in [4]).

A slot symmetry transformation s is a permutation acting on the slots $t \rightarrow T(\text{ind}[(g * s)[0]], \dots, \text{ind}[(g * s)[n - 1]])$

A dummy symmetry transformation acts on $\text{ind } t \rightarrow T(\text{ind}[(d * g)[0]], \dots, \text{ind}[(d * g)[n - 1]])$

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

$g \rightarrow d * g * s$, so it belongs to the coset $D * g * S$.

Let us explain the conventions by an example.

Given a tensor T^{d3d2d1}_{d1d2d3} with the slot symmetries

$$T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$$

$$T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$$

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering $d1, d2, d3$ then and contravariant index before covariant index; that is the canonical form of the tensor.

The canonical form is $-T^{d1d2d3}_{d1d2d3}$ obtained using $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$.

To convert this problem in the input for this function, use the following labelling of the index names (- for covariant for short) $d1, -d1, d2, -d2, d3, -d3$

T^{d3d2d1}_{d1d2d3} corresponds to $g = [4, 2, 0, 1, 3, 5, 6, 7]$ where the last two indices are for the sign $\text{sgens} = [\text{Permutation}(0, 2)(6, 7), \text{Permutation}(0, 4)(6, 7)]$

$\text{sgens}[0]$ is the slot symmetry $-(0, 2)$ $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$

$\text{sgens}[1]$ 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 $d1 \rightarrow -d1$
 $T^{d3d2d1}_{d1d2d3} = T^{d3d2}_{d1}^{d1}_{d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] \rightarrow [4, 2, 1, 0, 3, 5, 6, 7] =_a f_{\text{mul}}(d, g)$ which differs from $_a f_{\text{mul}}(g, d)$.

The slot symmetry acts from the right $s = [2, 1, 0, 3, 4, 5, 7, 6]$ exchanges slots 0 and 2 and changes sign $T^{d3d2d1}_{d1d2d3} = -T^{d1d2d3}_{d1d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] \rightarrow [0, 2, 4, 1, 3, 5, 7, 6] =_a f_{\text{mul}}(g, s)$

Example in which the tensor is zero, same slot symmetries as above: $T^{d3}_{d1, d2}^{d1}_{d3}^{d2}$

$= -T^{d3}_{d1, d3}^{d1}_{d2}^{d2}$ under slot symmetry $-(2, 4)$;

$= T^{d3d1}_{d3d1}^{d3d1}_{d2}^{d2}$ under slot symmetry $-(0, 2)$;

$= T^{d3}_{d1d3}^{d1}_{d2}^{d2}$ symmetric metric;

$= 0$ since two of these lines have tensors differ only for the sign.

The double coset $D * g * S$ consists of permutations $h = d * g * s$ corresponding to equivalent tensors; if there are two h which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to $[d1, -d1, d2, -d2, d3, -d3]$ that is $\text{rep} = \min(D * g * S) = \min([d * g * s \text{ for } d \text{ in } D \text{ for } s \text{ in } S])$

The indices are fixed one by one; first choose the lowest index for slot 0, then the lowest remaining index for slot 1, etc. Doing this one obtains a chain of stabilizers

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

where $[b0, b1, \dots] = \text{range}(b)$ is a base of the symmetric group; the strong base b_S of S is an ordered sublist of it; therefore it is sufficient to compute once the strong base generators of S using the Schreier-Sims algorithm; the stabilizers of the strong base generators are the strong base generators of the stabilizer subgroup.

$dbase = [p0, p1, \dots]$ is not in general in lexicographic order, so that one must recompute the strong base generators each time; however this is trivial, there is no need to use the Schreier-Sims algorithm for D .

The algorithm keeps a TAB of elements (s_i, d_i, h_i) where $h_i = d_i * g * s_i$ satisfying $h_i[j] = p_j$ for $0 \leq j < i$ starting from $s_0 = id, d_0 = id, h_0 = g$.

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

For $j < i$ $d_i * g * s_i * S_{b0, \dots, b_{i-1}} * b_j = D_{p0, \dots, p_{i-1}} * p_j$ so that for dx in $D_{p0, \dots, p_{i-1}}$ and sx in $S_{base[0], \dots, base[i-1]}$ one has $dx * d_i * g * s_i * sx * b_j = p_j$

Search for dx, sx such that this equation holds for $j = i$; it can be written as $s_i * sx * b_j = J, dx * d_i * g * J = p_j$ $sx * b_j = s_i * ** - 1 * J$; $sx = \text{trace}(s_i * ** - 1, S_{b0, \dots, b_{i-1}}) dx * ** - 1 * p_j = d_i * g * J$; $dx = \text{trace}(d_i * g * J, D_{p0, \dots, p_{i-1}})$

$s_{i+1} = s_i * \text{trace}(s_i * ** - 1 * J, S_{b0, \dots, b_{i-1}})$ $d_{i+1} = \text{trace}(d_i * g * J, D_{p0, \dots, p_{i-1}}) * ** - 1 * d_i$ $h_{i+1} * b_i = d_{i+1} * g * s_{i+1} * b_i = p_i$

$h_n * b_j = p_j$ for all j , so that h_n is the solution.

Add the found (s, d, h) to TAB1.

At the end of the iteration sort TAB1 with respect to the h ; if there are two consecutive h in TAB1 which differ only for the sign, the tensor is zero, so return 0; if there are two consecutive h which are equal, keep only one.

Then stabilize the slot generators under i and the dummy generators under p_i .

Assign $TAB = TAB1$ at the end of the iteration step.

At the end TAB contains a unique (s, d, h) , since all the slots of the tensor h have been fixed to have the minimum value according to the symmetries. The algorithm returns h .

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an i which does not belong to the slot base for which p_i is fixed by the dummy symmetry only, while i is not invariant from the slot stabilizer, so p_i is not in general the minimal value.

This algorithm differs slightly from the original algorithm [3]:

the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors h are eliminated from TAB.

Examples

```
>>> gens = [Permutation(x) for x in [[2, 1, 0, 3, 4, 5, 7, 6],
...                                  [4, 1, 2, 3, 0, 5, 7, 6]]]
>>> base = [0, 2]
>>> g = Permutation([4, 2, 0, 1, 3, 5, 6, 7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
[0, 1, 2, 3, 4, 5, 7, 6]
```

```
>>> g = Permutation([4, 1, 3, 0, 5, 2, 6, 7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
0
```

`diofant.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

```
>>> Permutation.print_cyclic = True
>>> get_symmetric_group_sgs(3)
([0, 1], [Permutation(4)(0, 1), Permutation(4)(1, 2)])
```

`diofant.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

```
>>> 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)])
```

4.4 Number Theory

Generating and counting primes.

class `diofant.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 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 $\leq n$ (a real number).

Examples

```
>>> 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 *i*th prime number.

i must be an integer.

The list is extended by 50% if it is too short, so it is likely that it will be longer than requested.

Examples

```
>>> from 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

```
>>> print(list(sieve.primerange(7, 18)))
[7, 11, 13, 17]
```

search(*n*)

Return the indices *i*, *j* of the primes that bound *n*.

If *n* is prime then *i* == *j*.

Although *n* can be an expression, if ceiling cannot convert it to an integer then an error will be raised.

Examples

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

`diofant.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.

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:

```
>>> def func(i):
...     return (i**2 + 1) % 51
```

and given a seed of 4 and the `mu` and `lambda` terms calculated:

```
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```
>>> n = cycle_length(func, 4, values=True)
>>> list(n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, `nmax` can be used to exit early (and `mu` will be returned as `None`):

```
>>> next(cycle_length(func, 4, nmax=4))
(4, None)
>>> list(cycle_length(func, 4, nmax=4, values=True))
[17, 35, 2, 5]
```

References

- https://en.wikipedia.org/wiki/Cycle_detection.

`diofant.ntheory.generate.nextprime(n, ith=1)`

Return the *ith* prime greater than *n*.

i must be an integer.

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> [(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
```

See also:

`prevprime` (page 231)

Return the largest prime smaller than *n*

`primerange` (page 232)

Generate all primes in a given range

`diofant.ntheory.generate.prevprime(n)`

Return the largest prime smaller than *n*.

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

See also:

`nextprime` (page 231)

Return the *ith* prime greater than *n*

`primerange` (page 232)

Generates all primes in a given range

`diofant.ntheory.generate.prime(nth)`

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

References

- <https://primes.utm.edu/glossary/xpage/BertrandsPostulate.html>

Examples

```
>>> prime(10)
29
>>> prime(1)
2
```

See also:

diofant.ntheory.primetest.isprime (page 251)

Test if n is prime

primerange (page 232)

Generate all primes in a given range

primepi (page 232)

Return the number of primes less than or equal to n

diofant.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.

Examples

```
>>> primepi(25)
9
```

See also:

diofant.ntheory.primetest.isprime (page 251)

Test if n is prime

primerange (page 232)

Generate all primes in a given range

prime (page 231)

Return the nth prime

diofant.ntheory.generate.primerange(a, b)

Generate a list of all prime numbers in the range [a, b).

If the range exists in the default sieve, the values will be returned from there; otherwise values will be returned but will not modify the sieve.

Notes

Some famous conjectures about the occurrence of primes in a given range are:

- **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 $\log(n)$; the gap between primes can be arbitrarily large since sequences of composite numbers are arbitrarily large, e.g. the numbers in the sequence $n! + 2, n! + 3 \dots n! + n$ are all composite.

References

- https://en.wikipedia.org/wiki/Prime_number
- <https://primes.utm.edu/notes/gaps.html>

Examples

```
>>> print(list(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]
```

See also:

***nextprime* (page 231)**

Return the *i*th prime greater than *n*

***prevprime* (page 231)**

Return the largest prime smaller than *n*

***randprime* (page 234)**

Returns a random prime in a given range

***primorial* (page 233)**

Returns the product of primes based on condition

***Sieve.primerange* (page 229)**

return range from already computed primes or extend the sieve to contain the requested range.

`diofant.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`).

```
>>> 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 232)

Generate all primes in a given range

`diofant.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`.

References

- https://en.wikipedia.org/wiki/Bertrand's_postulate

Examples

```
>>> randprime(1, 30)
29
>>> isprime(randprime(1, 30))
True
```

See also:

primerange (page 232)

Generate all primes in a given range

Integer factorization.

`diofant.ntheory.factor_.antidivisor_count(n)`

Return the number of antidivisors of *n*.

References

- formula from <https://oeis.org/A066272>

Examples

```
>>> antidivisor_count(13)
4
>>> antidivisor_count(27)
5
```

See also:

factorint (page 238), *divisors* (page 237), *antidivisors* (page 235), *divisor_count* (page 236), *totient* (page 246)

`diofant.ntheory.factor_.antidivisors(n, generator=False)`

Return all antidivisors of *n* sorted from 1..*n* by default.

Antidivisors of *n* are numbers that do not divide *n* by the largest possible margin. If *generator* is *True* an unordered generator is returned.

References

- definition is described in <http://oeis.org/A066272/a066272a.html>

Examples

```
>>> antidivisors(24)
[7, 16]
```

```
>>> sorted(antidivisors(128, generator=True))
[3, 5, 15, 17, 51, 85]
```

See also:

primefactors (page 244), *factorint* (page 238), *divisors* (page 237), *divisor_count* (page 236), *antidivisor_count* (page 235)

`diofant.ntheory.factor_.core(n, t=2)`

Calculate $\text{core}(n, t) = \text{core}_t(n)$ of a positive integer n .

$\text{core}_2(n)$ is equal to the squarefree part of n

If n 's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\text{core}_t(n) = \prod_{i=1}^{\omega} p_i^{m_i \bmod t}.$$

Parameters

t ($\text{core}(n, t)$ calculates the t -th power free part of n) – $\text{core}(n, 2)$ is the squarefree part of n $\text{core}(n, 3)$ is the cubefree part of n

Default for t is 2.

References

- https://en.wikipedia.org/wiki/Square-free_integer#Squarefree_core

Examples

```
>>> from diofant.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
379238
>>> core(15**11, 10)
15
```

See also:

`factorint` (page 238), `diofant.solvers.diophantine.square_factor` (page 633)

`diofant.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.

References

- <https://web.archive.org/web/20130629014824/http://www.mayer.dial.pipex.com:80/maths/formulae.htm>

Examples

```
>>> divisor_count(6)
4
```

See also:

[factorint](#) (page 238), [divisors](#) (page 237), [totient](#) (page 246)

class diofant.ntheory.factor_.**divisor_sigma**(n, k=1)

Calculate the divisor function $\sigma_k(n)$ for positive integer n.

divisor_sigma(n, k) is equal to $\sum(x^k \text{ for } x \text{ in } \text{divisors}(n))$

If n's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k(n) = \prod_{i=1}^{\omega} (1 + p_i^k + p_i^{2k} + \cdots + p_i^{m_i k}).$$

Parameters

k (power of divisors in the sum) - for k = 0, 1: divisor_sigma(n, 0) is equal to divisor_count(n) divisor_sigma(n, 1) is equal to sum(divisors(n))

Default for k is 1.

References

- https://en.wikipedia.org/wiki/Divisor_function

Examples

```
>>> divisor_sigma(18, 0)
6
>>> divisor_sigma(39, 1)
56
>>> divisor_sigma(12, 2)
210
>>> divisor_sigma(37)
38
```

See also:

[divisor_count](#) (page 236), [totient](#) (page 246), [divisors](#) (page 237), [factorint](#) (page 238)

diofant.ntheory.factor_.divisors(n, generator=False)

Return all divisors of n.

Divisors are 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).

Examples

```
>>> 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]
```

See also:

primefactors (page 244), *factorint* (page 238), *divisor_count* (page 236)

References

- <https://stackoverflow.com/questions/1010381/python-factorization>

`diofant.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:

```
>>> 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:

```
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f
{7: 1, 991: 1, 202916782076162456022877024859: 1}
>>> 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:

```
>>> pprint(factorint(4200, visual=True))
23·31·52·71
```

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`:

```
>>> regular = factorint(1764)
>>> regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular))
22·32·72
```

```
>>> visual = factorint(1764, visual=True)
>>> pprint(visual)
22·32·72
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:

Input	True	False	other
dict	mul	dict	mul
n	mul	dict	dict
mul	mul	dict	dict

Notes

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(f'{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.

See also:

[`smoothness`](#) (page 244), [`smoothness_p`](#) (page 245), [`divisors`](#) (page 237)

`diofant.ntheory.factor_.factorrat`(*rat*, *limit=None*, *use_trial=True*, *use_rho=True*, *use_pm1=True*, *verbose=False*, *visual=None*)

Given a Rational *r*, `factorrat(r)` returns a dict containing the prime factors of *r* as keys and their respective multiplicities as values. For example:

```
>>> factorrat(Rational(8, 9)) # = (2**3) * (3**(-2))
{2: 3, 3: -2}
>>> factorrat(Rational(-1, 987)) # = -1 * (3**(-1)) * (7**(-1)) * (47**(-1))
{-1: 1, 3: -1, 7: -1, 47: -1}
```

Please see the docstring for `factorint` for detailed explanations and examples of the following keywords:

- `limit`: Integer limit up to which trial division is done
- `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
- `verbose`: Toggle detailed printing of progress
- `visual`: Toggle product form of output

`diofant.ntheory.factor_.multiplicity`(*p*, *n*)

Find the greatest integer *m* such that p^m divides *n*.

Examples

```
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, Rational(1, 9))
-2
```

`diofant.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

```
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big=False)
(4, 2)
```

`diofant.ntheory.factor_.pollard_pm1(n, B=10, a=2, retries=0, seed=1234)`

Use Pollard's p-1 method to try to extract a nontrivial factor of n . Either a divisor (perhaps composite) or `None` is returned.

The value of a is the base that is used in the test $\gcd(a^{**M} - 1, n)$. The default is 2. If `retries > 0` then if no factor is found after the first attempt, a new a will be generated randomly (using the seed) and the process repeated.

Note: the value of M is $\text{lcm}(1..B) = \text{reduce}(\text{lcm}, \text{range}(2, B + 1))$.

A search is made for factors next to even numbers having a power smoothness less than B . Choosing a larger B increases the likelihood of finding a larger factor but takes longer. Whether a factor of n is found or not depends on a and the power smoothness of the even number just less than the factor p (hence the name $p - 1$).

Although some discussion of what constitutes a good a some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if $\gcd(a^{**M} - 1, n) = N$ then $a^{**M} \% q^{**r}$ is 1 for every prime power divisor of N . But consider the following:

```
>>> 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:

```
>>> M = 1
>>> for i in range(2, 256):
...     M = math.lcm(M, i)
>>> {math.gcd(pow(a, M, n) - 1, n) for a in range(2, 256) if
...   math.gcd(pow(a, M, n) - 1, n) != n}
{1009}
```

But does $a^M \% d$ for every divisor of n give 1?

```
>>> am = pow(255, M, n)
>>> [(d, am % Pow(*d.args)) for d in factorint(n, visual=True).args]
[(257**1, 1), (1009**1, 1)]
```

No, only one of them. So perhaps the principle is that a root will be found for a given value of B provided that:

- 1) the power smoothness of the $p - 1$ value next to the root does not exceed B
- 2) $a^*M \% p \neq 1$ for any of the divisors of n .

By trying more than one a it is possible that one of them will yield a factor.

Examples

With the default smoothness bound, this number can't be cracked:

```
>>> 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:

```
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B -pow are the same for the $p - 1$ factorizations of the divisors because those factorizations had a very large prime factor:

```
>>> factorint(4410317 - 1)
{2: 2, 617: 1, 1787: 1}
>>> factorint(4869863-1)
{2: 1, 2434931: 1}
```

Note that until B reaches the B -pow value of 1787, the number is not cracked;

```
>>> pollard_pm1(21477639576571, B=1786)
>>> pollard_pm1(21477639576571, B=1787)
4410317
```

The B value has to do with the factors of the number next to the divisor, not the divisors themselves. A worst case scenario is that the number next to the factor p has a large prime divisor or is a perfect power. If these conditions apply then the power-smoothness will be about $p/2$ or p . The more realistic is that there will be a large prime factor next to p requiring a B value on the order of $p/2$. Although primes may have been searched for up to this level, the $p/2$ is a factor of $p - 1$, something that we don't know. The modular.math reference below states that 15% of numbers in the range of 10^{15} to $15^{15} + 10^4$ are 10^6 power smooth so a B of 10^6 will fail 85% of the time in that range. From 10^8 to $10^8 + 10^3$ the percentages are nearly reversed...but in that range the simple trial division is quite fast.

References

- Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 236-238
- <https://web.archive.org/web/20150716201437/http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node81.html>
- <https://web.archive.org/web/20170830055619/http://www.cs.toronto.edu/~yuvalf/Factorization.pdf>

```
diofant.ntheory.factor_.pollard_rho(n, s=2, a=1, retries=5, seed=1234,
                                     max_steps=None, F=None)
```

Use Pollard's rho method to try to extract a nontrivial factor of n . The returned factor may be a composite number. If no factor is found, `None` is returned.

The algorithm generates pseudo-random values of x with a generator function, replacing x with $F(x)$. If F is not supplied then the function $x^2 + a$ is used. The first value supplied to $F(x)$ is s . Upon failure (if `retries` is > 0) a new a and s will be supplied; the a will be ignored if F was supplied.

The sequence of numbers generated by such functions generally have a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 - this leader and loop look a bit like the Greek letter rho, and thus the name, 'rho'.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> n = 16843009
>>> def f(x):
...     return (2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     a, b = next(cycle_length(f, s))
...     print(f'loop length = {a:4d}; leader length = {b:3d}')
loop length = 2489; leader length = 42
loop length = 78; leader length = 120
loop length = 1482; leader length = 99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x = 2
>>> for i in range(9):
...     x = (x**2 + 12) % 17
...     print(x)
16
13
11
14
4
11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12) % 17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12) % 17, 2, values=True))
[16, 13, 11, 14, 4]
```

Instead of checking the differences of all generated values for a gcd with n , only the k th and $2*k$ th numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If `max_steps` is specified, the iteration is cancelled with a failure after the specified number of steps.

Examples

```
>>> n = 16843009
>>> def f(x):
...     return (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
```

References

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 229-231

`diofant.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.

Examples

```
>>> primefactors(6)
[2, 3]
>>> primefactors(-5)
[5]
```

```
>>> sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]
>>> primefactors(123456)
[2, 3, 643]
```

```
>>> sorted(factorint(1000000001, limit=200).items())
[(101, 1), (99009901, 1)]
>>> isprime(99009901)
False
>>> primefactors(1000000001, limit=300)
[101]
```

See also:

[`divisors`](#) (page 237)

`diofant.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.


```
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

See also:

[factorint](#) (page 238), [smoothness_p](#) (page 245)

`diofant.ntheory.factor_.smoothness_p(n, m=-1, power=0, visual=None)`

Return a list of `[m, (p, (M, sm(p + m), psm(p + m)))]` where:

1. p^*M is the base- p divisor of n
2. $sm(p + m)$ is the smoothness of $p + m$ ($m = -1$ by default)
3. $psm(p + m)$ is the power smoothness of $p + m$

The list is sorted according to smoothness (default) or by power smoothness if $power=1$.

The smoothness of the numbers to the left ($m = -1$) or right ($m = 1$) of a factor govern the results that are obtained from the $p \pm 1$ type factoring methods.

```
>>> 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:

```
>>> f = factorint(17*9)
>>> f
{3: 2, 17: 1}
>>> smoothness_p(f)
'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:

Input	Visual		
	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 238), [smoothness](#) (page 244)

`diofant.ntheory.factor_.square_factor(a)`

Returns an integer c s.t. $a = c^2k$, $c, k \in \mathbb{Z}$. Here k is square free. a can be given as an integer or a dictionary of factors.

Examples

```
>>> square_factor(24)
2
>>> square_factor(-36*3)
6
>>> square_factor(1)
1
>>> square_factor({3: 2, 2: 1, -1: 1})
3
```

See also:

[`diofant.solvers.diophantine.reconstruct`](#) (page 634), [`diofant.ntheory.factor_.core`](#) (page 236)

class `diofant.ntheory.factor_.totient(n)`

Calculate the Euler totient function $\phi(n)$

```
>>> totient(1)
1
>>> totient(25)
20
```

See also:

[`divisor_count`](#) (page 236)

`diofant.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

```
>>> trailing(128)
7
>>> trailing(63)
0
```

`diofant.ntheory.modular.crt(M, U, 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 $f = u_i \bmod m_i$ for each pair out of U 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.

Examples

```
>>> 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)
```

Notes

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 248)

`diofant.ntheory.modular.crt1(M)`

First part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`diofant.ntheory.modular.crt2(M, U, p, E, S, symmetric=False)`

Second part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

`diofant.ntheory.modular.integer_rational_reconstruction(c, m, domain)`

Reconstruct a rational number $\frac{a}{b}$ from

$$c = \frac{a}{b} \bmod m,$$

where c and m are integers.

The algorithm is based on the Euclidean Algorithm. In general, m is not a prime number, so it is possible that b is not invertible modulo m . In that case `None` is returned.

Parameters

- **c** (*Integer*) - $c = \frac{a}{b} \bmod m$
- **m** (*Integer*) - modulus, not necessarily prime
- **domain** (*IntegerRing*) - a, b, c are elements of domain

Returns

frac (*Rational*) - either $\frac{a}{b}$ in \mathbb{Q} or `None`

References

- [Wan81]

`diofant.ntheory.modular.solve_congruence(*remainder_modulus_pairs, **hint)`

Compute the integer n that has the residual a_i when it is divided by m_i where the a_i and m_i are given as pairs to this function: $((a_1, m_1), (a_2, m_2), \dots)$. If there is no solution, return. Otherwise return n and its modulus.

The m_i values need not be co-prime. If it is known that the moduli are not co-prime then the hint check can be set to `False` (default=`True`) and the check for a quicker solution via `crt()` (valid when the moduli are co-prime) will be skipped.

If the hint `symmetric` is `True` (default is `False`), the value of n will be within $1/2$ of the modulus, possibly negative.

Examples

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)
```

See also:

crt (page 246)

high level routine implementing the Chinese Remainder Theorem

`diofant.ntheory.modular.symmetric_residue(a, m)`

Return the residual mod m such that it is within half of the modulus.

```
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

`diofant.ntheory.multinomial.binomial_coefficients(n)`

Return a dictionary containing pairs $(k_1, k_2) : C_{kn}$ where C_{kn} are binomial coefficients and $n = k_1 + k_2$.

Examples

```
>>> 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 249), [multinomial_coefficients](#) (page 249)

`diofant.ntheory.multinomial.binomial_coefficients_list(n)`

Return a list of binomial coefficients as rows of the Pascal's triangle.

Examples

```
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

See also:

[binomial_coefficients](#) (page 249), [multinomial_coefficients](#) (page 249)

`diofant.ntheory.multinomial.multinomial_coefficients(m, n)`

Return a dictionary containing pairs $\{(k_1, k_2, \dots, k_m) : C_{kn}\}$ where C_{kn} are multinomial coefficients such that $n = k_1 + k_2 + \dots + k_m$.

Examples

```
>>> multinomial_coefficients(2, 5)
{(0, 5): 1, (1, 4): 5,
 (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
```

Notes

The algorithm is based on the following result:

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

See also:

[`binomial_coefficients_list`](#) (page 249), [`binomial_coefficients`](#) (page 249)

`diofant.ntheory.multinomial.multinomial_coefficients_iterator(m, n,`
`_tuple=<class 'tuple'>)`

Multinomial coefficient iterator.

This routine has been optimized for m large with respect to n by taking advantage of the fact that when the monomial tuples t are stripped of zeros, their coefficient is the same as that of the monomial tuples from `multinomial_coefficients(n, n)`. Therefore, the latter coefficients are precomputed to save memory and time.

```
>>> 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

```
>>> it = multinomial_coefficients_iterator(20, 3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1)
```

`diofant.ntheory.partitions_.npartitions(n)`

Give the number $P(n)$ of unrestricted partitions of the integer n .

The partition function $P(n)$ computes the number of ways that n can be written as a sum of positive integers, where the order of addends is not considered significant.

Examples

```
>>> npartitions(25)
1958
```

References

- <https://mathworld.wolfram.com/PartitionFunctionP.html>

Primality testing.

`diofant.ntheory.primetest.is_square(n, prep=True)`

Return True if $n == a * a$ for some integer a , else False.

If n is suspected of *not* being a square then this is a quick method of confirming that it is not.

References

- <https://mersenneforum.org/showpost.php?p=110896>

See also:

diofant.core.power.integer_nthroot (page 101)

`diofant.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 2×10^{-28} .

Examples

```
>>> isprime(13)
True
>>> isprime(15)
False
```

See also:

diofant.ntheory.generate.primerange (page 232)

Generates all primes in a given range

diofant.ntheory.generate.primepi (page 232)

Return the number of primes less than or equal to n

diofant.ntheory.generate.prime (page 231)

Return the n th prime

`diofant.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: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants

Examples

```
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

`diofant.ntheory.residue_ntheory.discrete_log(n, a, b, order=None, prime_order=None)`

Compute the discrete logarithm of a to the base b modulo n.

This is a recursive function to reduce the discrete logarithm problem in cyclic groups of composite order to the problem in cyclic groups of prime order.

Notes

It employs different algorithms depending on the problem (subgroup order size, prime order or not):

- Trial multiplication
- Baby-step giant-step
- Pohlig-Hellman

References

- <https://mathworld.wolfram.com/DiscreteLogarithm.html>
- [MVV97]

Examples

```
>>> discrete_log(41, 15, 7)
3
```

`diofant.ntheory.residue_ntheory.is_nthpow_residue(a, n, m)`

Returns True if $x^n \equiv a \pmod{m}$ has solutions.

References

- P. Hackman “Elementary Number Theory” (2009), page 76

`diofant.ntheory.residue_ntheory.is_primitive_root(a, p)`

Returns True if a is a primitive root of p

a is said to be the primitive root of p if $\gcd(a, p) = 1$ and $\text{totient}(p)$ is the smallest positive number s.t.:

```
a**totient(p) cong 1 mod(p)
```


Examples

```
>>> 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
```

`diofant.ntheory.residue_ntheory.is_quad_residue(a, p)`

Returns True if $a \pmod{p}$ is in the set of squares mod p , i.e. $a \equiv i^2 \pmod{p}$ for i in $\text{range}(p)$. If p is an odd prime, an iterative method is used to make the determination:

```
>>> sorted({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 254), [jacobi_symbol](#) (page 253)

`diofant.ntheory.residue_ntheory.jacobi_symbol(m, n)`

Returns the Jacobi symbol (m/n) .

For any integer m and any positive odd integer n the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n :

$$\left(\frac{m}{n}\right) = \left(\frac{m}{p^1}\right)^{\alpha_1} \left(\frac{m}{p^2}\right)^{\alpha_2} \cdots \left(\frac{m}{p^k}\right)^{\alpha_k} \text{ where } n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

Like the Legendre symbol, if the Jacobi symbol $(\frac{m}{n}) = -1$ then m is a quadratic nonresidue modulo n .

But, unlike the Legendre symbol, if the Jacobi symbol $(\frac{m}{n}) = 1$ then m may or may not be a quadratic residue modulo n .

Parameters

- **m** (*integer*)
- **n** (*odd positive integer*)

Examples

```
>>> 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
>>> Integer(45).factors()
{3: 2, 5: 1}
>>> jacobi_symbol(7, 45) == L(7, 3)**2 * L(7, 5)**1
True
```

See also:

[is_quad_residue](#) (page 253), [legendre_symbol](#) (page 254)

`diofant.ntheory.residue_ntheory.legendre_symbol(a, p)`

Returns the Legendre symbol (a/p) .

For an integer a and an odd prime p , the Legendre symbol is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \text{ divides } a \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \end{cases}$$

Parameters

- **a** (*integer*)
- **p** (*odd prime*)

Examples

```
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, -1, -1, -1]
>>> sorted({i**2 % 7 for i in range(7)})
[0, 1, 2, 4]
```

See also:

[*is_quad_residue*](#) (page 253), [*jacobi_symbol*](#) (page 253)

References

- https://en.wikipedia.org/wiki/Legendre_symbol

class `diofant.ntheory.residue_ntheory.mobius(n)`

Möbius function maps natural number to $\{-1, 0, 1\}$

It is defined as follows:

- 1) 1 if $n = 1$.
- 2) 0 if n has a squared prime factor.
- 3) $(-1)^k$ if n is a square-free positive integer with k number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Möbius Function model).

Parameters

- **n** (*positive integer*)

Examples

```
>>> mobius(13*7)
1
>>> mobius(1)
1
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```

References

- https://en.wikipedia.org/wiki/M%C3%B6bius_function
- Thomas Koshy “Elementary Number Theory with Applications”

`diofant.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

```
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

`diofant.ntheory.residue_ntheory.nthroot_mod(a, n, p, all_roots=False)`

Find the solutions to $x^n = a \pmod{p}$.

Parameters

- **a** (*integer*)
- **n** (*positive integer*)
- **p** (*positive integer*)
- **all_roots** (*if False returns the smallest root, else the list of roots*)

Examples

```
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

`diofant.ntheory.residue_ntheory.primitive_root(p)`

Returns the smallest primitive root or None.

References

- W. Stein “Elementary Number Theory” (2011), page 44
- P. Hackman “Elementary Number Theory” (2009), Chapter C

Parameters

p (*positive integer*)

Examples

```
>>> primitive_root(19)
2
```

`diofant.ntheory.residue_ntheory.quadratic_residues(p)`

Returns the list of quadratic residues.

Examples

```
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

`diofant.ntheory.residue_ntheory.sqrt_mod(a, p, all_roots=False)`

Find a root of $x^2 = a \pmod p$.

Parameters

- **a** (*integer*)
- **p** (*positive integer*)
- **all_roots** (*if True the list of roots is returned or None*)

Notes

If there is no root it is returned `None`; else the returned root is less or equal to $p // 2$; in general is not the smallest one. It is returned $p // 2$ only if it is the only root.

Use `all_roots` only when it is expected that all the roots fit in memory; otherwise use `sqrt_mod_iter`.

Examples

```
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

`diofant.ntheory.residue_ntheory.sqrt_mod_iter(a, p, domain=<class 'int'>)`

Iterate over solutions to $x^2 = a \pmod p$.

Parameters

- **a** (*integer*)

- **p** (*positive integer*)
- **domain** (integer domain, int, ZZ or Integer)

Examples

```
>>> list(sqrt_mod_iter(11, 43))
[21, 22]
```

`diofant.ntheory.continued_fraction.continued_fraction_convergents(cf)`

Return an iterator over the convergents of a continued fraction.

The parameter should be an iterable returning successive partial quotients of the continued fraction, such as might be returned by `continued_fraction_iterator`. In computing the convergents, the continued fraction need not be strictly in canonical form (all integers, all but the first positive). Rational and negative elements may be present in the expansion.

Examples

```
>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]
```

```
>>> list(continued_fraction_convergents([1, Rational(1, 2), -7, Rational(1, 4)]))
[1, 3, 19/5, 7]
```

```
>>> it = continued_fraction_convergents(continued_fraction_iterator(pi))
>>> for n in range(7):
...     print(next(it))
3
22/7
333/106
355/113
103993/33102
104348/33215
208341/66317
```

See also:

[`continued_fraction_iterator`](#) (page 257)

`diofant.ntheory.continued_fraction.continued_fraction_iterator(x)`

Return continued fraction expansion of `x` as iterator.

Examples

```
>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]
>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 1, 2]
```

```
>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
```

(continues on next page)

(continued from previous page)

```
292
+
+
1
```

References

- https://en.wikipedia.org/wiki/Continued_fraction

`diofant.ntheory.continued_fraction.continued_fraction_periodic(p, q, d=0)`

Find the periodic continued fraction expansion.

Compute the continued fraction expansion of a rational or a quadratic surd, i.e. $\frac{p+\sqrt{d}}{q}$, where p, q and $d \geq 0$ are integers.

Returns

list – the continued fraction representation (canonical form) as a list of integers, optionally ending (for quadratic irrationals) with repeating block as the last term of this list.

Parameters

- **p** (*int*) – the rational part of the number’s numerator
- **q** (*int*) – the denominator of the number
- **d** (*int, optional*) – the irrational part (discriminator) of the number’s numerator

Examples

```
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, -1]]
```

Golden ratio has the simplest continued fraction expansion:

```
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

See also:

[continued_fraction_iterator](#) (page 257), [continued_fraction_reduce](#) (page 259)

References

- https://en.wikipedia.org/wiki/Periodic_continued_fraction
- K. Rosen. Elementary Number theory and its applications. Addison-Wesley, 3 Sub edition, pages 379-381, January 1992.

`diofant.ntheory.continued_fraction.continued_fraction_reduce(cf)`

Reduce a continued fraction to a rational or quadratic irrational.

Compute the rational or quadratic irrational number from its terminating or periodic continued fraction expansion. The continued fraction expansion (cf) should be supplied as a terminating iterator supplying the terms of the expansion. For terminating continued fractions, this is equivalent to `list(continued_fraction_convergents(cf))[-1]`, only a little more efficient. If the expansion has a repeating part, a list of the repeating terms should be returned as the last element from the iterator. This is the format returned by `continued_fraction_periodic`.

For quadratic irrationals, returns the largest solution found, which is generally the one sought, if the fraction is in canonical form (all terms positive except possibly the first).

Examples

```
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]).evalf(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
1/2 + sqrt(5)/2
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```

See also:

`continued_fraction_periodic` (page 258)

`diofant.ntheory.egyptian_fraction.egyptian_fraction(r, algorithm='Greedy')`

Compute an Egyptian fraction of the rational r .

Returns

list – The list of denominators of an Egyptian fraction expansion.

Parameters

- **r** (*Rational*) – a positive rational number.
- **algorithm** ({ “Greedy”, “Graham Jewett”, “Takenouchi”, “Golomb” }, *optional*) – Denotes the algorithm to be used (the default is “Greedy”).

Examples

```
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction(Rational(3, 7), 'Graham Jewett')
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction(Rational(3, 7), 'Takenouchi')
[4, 7, 28]
>>> egyptian_fraction(Rational(3, 7), 'Golomb')
[3, 15, 35]
>>> egyptian_fraction(Rational(11, 5), 'Golomb')
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

See also:

[*diofant.core.numbers.Rational*](#) (page 88)

Notes

Currently the following algorithms are supported:

1) Greedy Algorithm

Also called the Fibonacci-Sylvester algorithm. At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

It has some distinct properties:

- a) Given p/q in lowest terms, generates an expansion of maximum length p . Even as the numerators get large, the number of terms is seldom more than a handful.
- b) Uses minimal memory.
- c) The terms can blow up (standard examples of this are $5/121$ and $31/311$). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

2) Graham Jewett Algorithm

The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always $2^{(x/\gcd(x, y)) - 1}$.

3) Takenouchi Algorithm

The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates.

4) Golomb's Algorithm

A method given by Golomb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972).

If the given rational is greater than or equal to 1, a greedy algorithm of summing the harmonic sequence $1/1 + 1/2 + 1/3 + \dots$ is used, taking all the unit fractions of this sequence until adding one more would be greater than the given number. This list of denominators is prefixed to the result from the requested algorithm used on the remainder. For example, if r is $8/3$, using the Greedy algorithm, we get $[1, 2, 3, 4, 5, 6, 7, 14, 420]$, where the beginning of the sequence, $[1, 2, 3, 4, 5, 6, 7]$ is part of the harmonic sequence summing to $363/140$, leaving a remainder of $31/420$, which yields $[14, 420]$ by the Greedy algorithm. The result of `egyptian_fraction(Rational(8, 3), "Golomb")` is $[1, 2, 3, 4, 5, 6, 7, 14, 574, 2788, 6460, 11590, 33062, 113820]$, and so on.

References

- https://en.wikipedia.org/wiki/Egyptian_fraction
- https://en.wikipedia.org/wiki/Greedy_algorithm_for_Egyptian_fractions
- <https://www.ics.uci.edu/~eppstein/numth/egypt/conflict.html>
- https://ami.uni-eszterhazy.hu/uploads/papers/finalpdf/AMI_42_from129to134.pdf

4.5 Concrete Mathematics

4.5.1 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:

```
>>> (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)
```

(continues on next page)

(continued from previous page)

```
True
>>> (2**(n-7)).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise it will return `None` to say that sequence is not hypergeometric:

```
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

4.5.2 Concrete Class Reference

class `diofant.concrete.summations.Sum(function, *symbols, **assumptions)`

Represents an unevaluated summation.

`Sum` represents a finite or infinite series, with the first argument being the general form of terms in the series (which usually depend on the bound variable `symbol`), and the second argument being `(symbol, start, end)`, with `symbol` taking all integer values from `start` through `end` (inclusive).

For `start < end` we adopt definition:

```
Sum(f(i), (i, start, end)) = -Sum(f(i), (i, end+1, start-1))
```

Notes

The summation convention for `start < end` described by Karr in [Kar81], especially definition 3 of section 1.4. The only difference with the reference is that Karr defines all sums with the upper limit being exclusive. This is in contrast to the usual mathematical notation, which we adopt, but does not affect the summation convention. Indeed we have:

$$\sum_{m \leq i < n} f_i = \sum_{i=m}^{n-1} f_i$$

This convention allows us to preserve the splitting identity

$$\sum_{i=m}^n f_i = \sum_{i=m}^l f_i + \sum_{i=l+1}^n f_i$$

regardless of the ordering of m , l and n .

Note that it also follows:

$$\sum_{i=m}^{m-1} f_i = 0$$

Examples

```
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()
E**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 adopted rule:

```
>>> Sum(k, (k, 1, n)).doit()
n**2/2 + n/2
>>> _.subs({n: -4})
6
>>> Sum(-n, (n, -3, 0)).doit()
6
```

See also:

summation (page 271), *diofant.concrete.products.Product* (page 265), *diofant.concrete.products.product* (page 272)

References

- <https://en.wikipedia.org/wiki/Summation>

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):

```
>>> Sum(1/k, (k, 2, 5)).doit().evalf()
1.2833333333333333
>>> s, e = Sum(1/k, (k, 2, 5)).euler_maclaurin()
>>> s
-log(2) + 7/20 + log(5)
>>> 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.

findrecur($F=Function('F')$, $n=None$)

Find a recurrence formula for the summand of the sum.

Given a sum $f(n) = \sum_k F(n, k)$, where $F(n, k)$ is doubly hypergeometric (that's, both $F(n+1, k)/F(n, k)$ and $F(n, k+1)/F(n, k)$ are rational functions of n and k), we find a recurrence for the summand $F(n, k)$ of the form

$$\sum_{i=0}^I \sum_{j=0}^J a_{i,j} F(n-j, k-i) = 0$$

Examples

```
>>> s = Sum(factorial(n)/(factorial(k)*factorial(n - k)), (k, 0, oo))
>>> s.findrecur()
-F(n, k) + F(n - 1, k) + F(n - 1, k - 1)
```

Notes

We use Sister Celine's algorithm, see [PetkovsekWZ97], Ch. 4.

reverse_order(*indices)

Reverse the order of a limit in a Sum.

Parameters

***indices** (*list*) - The selectors in the argument indices specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```
>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> s = Sum(x**2, (x, a, b), (x, c, d))
>>> Sum(x**2, (x, a, b), (x, c, d))
>>> s0 = s.reverse_order(0)
>>> s0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> s1 = s0.reverse_order(1)
>>> s1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 270),
[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#)
(page 271), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 270)

References

- [Kar81]

class `diofant.concrete.products.Product`(*function*, **symbols*, ***assumptions*)

Represents an unevaluated products.

Product represents a finite or infinite product, with the first argument being the general form of terms in the series (which usually depend on the bound variable `symbol`), and the second argument being (`symbol`, `start`, `end`), with `symbol` taking all integer values from `start` through `end` (inclusive).

Notes

We follow the the analogue of the summation convention described by Karr [Kar81], adopted by the [Sum](#) (page 262):

$$\prod_{i=m}^n f_i = \frac{1}{\prod_{i=n+1}^{m-1} f_i}$$

Examples

```
>>> Product(k**2, (k, 1, m)).doit()
factorial(m)**2
```

Products with the lower limit being larger than the upper one:

```
>>> Product(1/k, (k, 6, 1)).doit()
120
>>> Product(k, (k, 2, 5)).doit()
120
```

The empty product:

```
>>> Product(k, (k, n, n-1)).doit()
1
```

See also:

[diofant.concrete.summations.Sum](#) (page 262), [diofant.concrete.summations.summation](#) (page 271), [product](#) (page 272)

References

- https://en.wikipedia.org/wiki/Multiplication#Capital_pi_notation
- https://en.wikipedia.org/wiki/Empty_product

`reverse_order(*indices)`

Reverse the order of a limit in a Product.

Parameters

***indices** (*list*) - The selectors in the argument indices specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr)
gamma(b + 1)/gamma(a)
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P)
gamma(b + 1)/gamma(a)
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> s = Sum(x*y, (x, a, b), (y, c, d))
>>> s
Sum(x*y, (x, a, b), (y, c, d))
>>> s0 = s.reverse_order(0)
>>> s0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> s1 = s0.reverse_order(1)
>>> s1
Sum(x*y, (x, b + 1, a - 1), (y, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

`diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index` (page 270), `diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit` (page 271), `diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder` (page 270)

References

- [Kar81]

class `diofant.concrete.expr_with_limits.ExprWithLimits`(*function*, **symbols*, ***assumptions*)

Represents an expression with limits.

as_dummy()

Replace instances of the given dummy variables with explicit dummy counterparts to make clear what are dummy variables and what are real-world symbols in an object.

Examples

```
>>> Integral(x, (x, x, y), (y, x, y)).as_dummy()
Integral(_x, (_x, x, _y), (_y, x, y))
```

If the object supports the “integral at” limit (*x*,) it is not treated as a dummy, but the explicit form, (*x*, *x*) of length 2 does treat the variable as a dummy.

```
>>> Integral(x, x).as_dummy()
Integral(x, x)
>>> Integral(x, (x, x)).as_dummy()
Integral(_x, (_x, x))
```

If there were no dummies in the original expression, then the the symbols which cannot be changed by `subs()` are clearly seen as those with an underscore prefix.

See also:

[*diofant.concrete.expr_with_limits.ExprWithLimits.variables*](#) (page 268)

Lists the integration variables

property `free_symbols`

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> Sum(x, (x, y, 1)).free_symbols
{y}
```

property `function`

Return the function applied across limits.

Examples

```
>>> Integral(x**2, x).function
x**2
```

See also:

[diofant.concrete.expr_with_limits.ExprWithLimits.limits](#) (page 268),
[diofant.concrete.expr_with_limits.ExprWithLimits.variables](#) (page 268),
[diofant.concrete.expr_with_limits.ExprWithLimits.free_symbols](#)
(page 267)

property `is_number`

Return True if the Sum has no free symbols, else False.

property `limits`

Return the limits of expression.

Examples

```
>>> from diofant.abc import i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)
```

See also:

[diofant.concrete.expr_with_limits.ExprWithLimits.function](#) (page 267),
[diofant.concrete.expr_with_limits.ExprWithLimits.variables](#) (page 268),
[diofant.concrete.expr_with_limits.ExprWithLimits.free_symbols](#)
(page 267)

property `variables`

Return a list of the dummy variables

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

See also:

[diofant.concrete.expr_with_limits.ExprWithLimits.function](#) (page 267),
[diofant.concrete.expr_with_limits.ExprWithLimits.limits](#) (page 268),
[diofant.concrete.expr_with_limits.ExprWithLimits.free_symbols](#)
(page 267)

[diofant.concrete.expr_with_limits.ExprWithLimits.as_dummy](#) (page 267)
Rename dummy variables

class `diofant.concrete.expr_with_intlimits.ExprWithIntLimits`(*function*, **symbols*,
***assumptions*)

Represents an expression with integer limits.

`change_index`(*var*, *trafo*, *newvar*=None)

Change index of a Sum or Product.

Perform a linear transformation $x \mapsto ax + b$ on the index variable x . For a the only values allowed are ± 1 . A new variable to be used after the change of index can also be specified.

Parameters

- **var** (*Symbol*) – specifies the index variable x to transform.
- **trafo** (*Expr*) – The linear transformation in terms of var.
- **newvar** (*Symbol, optional*) – Replacement symbol to be used instead of var in the final expression.

Examples

```
>>> from diofant.abc import i, j, l, u, v
```

```
>>> s = Sum(x, (x, a, b))
>>> s.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> sn = s.change_index(x, x + 1, y)
>>> sn
Sum(y - 1, (y, a + 1, b + 1))
>>> sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> sn = s.change_index(x, -x, y)
>>> sn
Sum(-y, (y, -b, -a))
>>> sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> sn = s.change_index(x, x+u)
>>> sn
Sum(-u + x, (x, a + u, b + u))
>>> sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> sn = s.change_index(x, -x - u, y)
>>> sn
Sum(-u - y, (y, -b - u, -a - u))
>>> sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2
```

```
>>> p = Product(i*j**2, (i, a, b), (j, c, d))
>>> p
Product(i*j**2, (i, a, b), (j, c, d))
>>> p2 = p.change_index(i, i+3, k)
>>> p2
Product(j**2*(k - 3), (k, a + 3, b + 3), (j, c, d))
>>> p3 = p2.change_index(j, -j, l)
>>> p3
Product(l**2*(k - 3), (k, a + 3, b + 3), (l, -d, -c))
```

When dealing with symbols only, we can make a general linear transformation:

```
>>> sn = s.change_index(x, u*x+v, y)
>>> sn
Sum((-v + y)/u, (y, b*u + v, a*u + v))
>>> sn.doit()
-v*(a*u - b*u + 1)/u + (a**2*u**2/2 + a*u*v + a*u/2 - b**2*u**2/2 - b*u*v +
↳ b*u/2 + v)/u
>>> simplify(sn.doit())
a**2*u/2 + a/2 - b**2*u/2 + b/2
```

However, the last result can be inconsistent with usual summation where the index increment is always 1. This is obvious as we get back the original value only for u equal $+1$ or -1 .

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 270), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#) (page 271), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 270), [diofant.concrete.summations.Sum.reverse_order](#) (page 264), [diofant.concrete.products.Product.reverse_order](#) (page 266)

index(x)

Return the index of a dummy variable in the list of limits.

Note that we start counting with 0 at the inner-most limits tuple.

Parameters

x (*Symbol*) – a dummy variable

Examples

```
>>> Sum(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Sum(x*y, (x, a, b), (y, c, d)).index(y)
1
>>> Product(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Product(x*y, (x, a, b), (y, c, d)).index(y)
1
```

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#) (page 271), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 270), [diofant.concrete.summations.Sum.reverse_order](#) (page 264), [diofant.concrete.products.Product.reverse_order](#) (page 266)

reorder(* arg)

Reorder limits in a expression containing a Sum or a Product.

Parameters

*** arg** (*list of tuples*) – These tuples can contain numerical indices or index variable names or involve both.

Examples

```
>>> from diofant.abc import e, f
```

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((x, y))
Sum(x*y, (y, c, d), (x, a, b))
```

```
>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder((x, y), (x, z), (y,
←z))
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

```
>>> P = Product(x*y*z, (x, a, b), (y, c, d), (z, e, f))
>>> P.reorder((x, y), (x, z), (y, z))
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

We can also select the index variables by counting them, starting with the inner-most one:

```
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder((0, 1))
Sum(x**2, (x, c, d), (x, a, b))
```

And of course we can mix both schemes:

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, x))
Sum(x*y, (y, c, d), (x, a, b))
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, 0))
Sum(x*y, (y, c, d), (x, a, b))
```

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 270),
[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#)
 (page 271), [diofant.concrete.summations.Sum.reverse_order](#) (page 264),
[diofant.concrete.products.Product.reverse_order](#) (page 266)

reorder_limit(x, y)

Interchange two limit tuples of a Sum or Product expression.

Parameters

x, y (*int*) – are integers corresponding to the index variables of the two limits which are to be interchanged.

Examples

```
>>> from diofant.abc import e, f
```

```
>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder_limit(1, 0)
Sum(x**2, (x, c, d), (x, a, b))
```

```
>>> Product(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 270),
[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#)
 (page 270), [diofant.concrete.summations.Sum.reverse_order](#) (page 264),
[diofant.concrete.products.Product.reverse_order](#) (page 266)

4.5.3 Concrete Functions Reference

diofant.concrete.summations.summation(f, *symbols, **kwargs)

Compute the summation of f with respect to symbols.

The notation for symbols is similar to the notation used in Integral. `summation(f, (i, a, b))` computes the sum of f with respect to i from a to b, i.e.,

$$\text{summation}(f, (i, a, b)) = \sum_{i=a}^b f$$

If it cannot compute the sum, it returns an unevaluated Sum object. Repeated sums can be computed by introducing additional symbols tuples:

```
>>> i = symbols('i', 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
```

```
>>> summation(x**n/factorial(n), (n, 0, oo))
E**x
```

See also:

[diofant.concrete.summations.Sum](#) (page 262), [diofant.concrete.products.Product](#) (page 265), [diofant.concrete.products.product](#) (page 272)

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

Compute the product.

The notation for symbols is similar to the notation used in Sum or Integral. `product(f, (i, a, b))` computes the product of `f` with respect to `i` from `a` to `b`, i.e.,

$$\text{product}(f(n), (i, a, b)) = \prod_{i=a}^b f(n)$$

If it cannot compute the product, it returns an unevaluated Product object. Repeated products can be computed by introducing additional symbols tuples:

```
>>> i = symbols('i', 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))
```

diofant.concrete.gosper.gosper_normal(f, g, n)

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

```
>>> gosper_normal(4*n + 5, 2*(4*n + 1)*(2*n + 3), n)
(Poly(1/4, n, domain='QQ'), Poly(n + 3/2, n, domain='QQ'),
 Poly(n + 1/4, n, domain='QQ'))
```

`diofant.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

```
>>> gosper_term((4*n + 1)*factorial(n)/factorial(2*n + 1), n)
(-n - 1/2)7(n + 1/4)
```

`diofant.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.

Examples

```
>>> gosper_sum((4*k + 1)*factorial(k)/factorial(2*k + 1), (k, 0, n))
(-factorial(n) + 2*factorial(2*n + 1))/factorial(2*n + 1)
```

References

- [PetkovvsekWZ97]

4.6 Mathematical Functions

All functions support the methods documented below, inherited from *diofant.core.function.Function* (page 126).

class diofant.core.function.**Function**(*args)

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:

```
>>> g = 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 MyFunc that represents a mathematical function *MyFunc*. Suppose that it is well known, that *MyFunc*(0) is 1 and *MyFunc* at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that *MyFunc*(x) is real exactly when x is real. Here is an implementation that honours those requirements:

```
>>> class MyFunc(Function):
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x == 0:
...                 return Integer(1)
...             elif x is oo:
...                 return Integer(0)
...         def _eval_is_real(self):
...             return self.args[0].is_real
>>> MyFunc(0) + sin(0)
1
>>> MyFunc(oo)
0
>>> MyFunc(3.54).evalf() # Not yet implemented for MyFunc.
MyFunc(3.54)
>>> MyFunc(I).is_real
False
```

In order for MyFunc to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then nargs must be defined, e.g. if MyFunc can take one or two arguments then,

```
>>> class MyFunc(Function):
>>>     nargs = (1, 2)
>>>
```

classmethod class_key()

Nice order of classes.

fdiff(*argindex*=1)

Returns the first derivative of the function.

4.6.1 Elementary

This module implements elementary functions such as trigonometric, hyperbolic as well as functions like Abs, Max, sqrt etc.

4.6.2 diofant.functions.elementary.complexes

re

class diofant.functions.elementary.complexes.**re**(*arg*)

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.

Examples

```
>>> re(2*E)
2*E
>>> re(2*I + 17)
17
>>> re(2*I)
0
>>> re(im(x) + x*I + 2)
2
```

See also:

[*diofant.functions.elementary.complexes.im*](#) (page 275)

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

Returns the real number with a zero imaginary part.

im

class diofant.functions.elementary.complexes.**im**(*arg*)

Returns imaginary part of expression.

This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use Basic.as_real_imag() or perform complex expansion on instance of this function.

Examples

```
>>> im(2*I)
0
>>> re(2*I + 17)
17
>>> im(x*I)
re(x)
>>> im(re(x) + y)
im(y)
```

See also:

[*diofant.functions.elementary.complexes.re*](#) (page 275)

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

Return the imaginary part with a zero real part.

Examples

```
>>> im(2 + 3*I).as_real_imag()
(3, 0)
```

sign

class `diofant.functions.elementary.complexes.sign`(*arg*)

Returns the complex sign of an expression.

For nonzero complex number z is an equivalent of $z/abs(z)$. Else returns zero.

Examples

```
>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> .evalf()
0.707106781186548 + 0.707106781186548*I
```

See also:

[*Abs*](#) (page 276), [*conjugate*](#) (page 278)

Abs

class `diofant.functions.elementary.complexes.Abs`(*arg*)

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 Diofant expression to the built-in `abs()`, it will pass it automatically to `Abs()`.

Examples

```
>>> Abs(-1)
1
>>> x = Symbol('x', real=True)
>>> abs(-x) # The Python built-in
Abs(x)
>>> abs(x**2)
x**2
```

Note that the Python built-in will return either an Expr or int depending on the argument:

```
>>> type(abs(-1))
<...int'>
>>> type(abs(Integer(-1)))
<class 'diofant.core.numbers.One'>
```

Abs will always return a diofant object.

See also:

[diofant.functions.elementary.complexes.sign](#) (page 276), [diofant.functions.elementary.complexes.conjugate](#) (page 278)

fdiff(argindex=1)

Get the first derivative of the argument to Abs().

Examples

```
>>> abs(-x).fdiff()
sign(x)
```

adjoint

class diofant.functions.elementary.complexes.**adjoint**(arg)

Conjugate transpose or Hermite conjugation.

arg

class diofant.functions.elementary.complexes.**arg**(arg)

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

For a real number, the argument is always 0.

Examples

```
>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
```

conjugate

class diofant.functions.elementary.complexes.**conjugate**(arg)

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

```
>>> conjugate(2)
2
>>> conjugate(I)
-I
```

See also:

diofant.functions.elementary.complexes.sign (page 276), *diofant.functions.elementary.complexes.Abs* (page 276)

References

- https://en.wikipedia.org/wiki/Complex_conjugation

polar_lift

class diofant.functions.elementary.complexes.**polar_lift**(arg)

Lift argument to the Riemann surface of the logarithm, using the standard branch.

```
>>> p = Symbol('p', polar=True)
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)
```

```
>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

diofant.functions.elementary.exponential.exp_polar (page 295), *diofant.functions.elementary.complexes.periodic_argument* (page 279)

periodic_argument

class diofant.functions.elementary.complexes.**periodic_argument**(*ar, period*)

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period P , always return a value in $(-P/2, P/2]$, by using $\exp(P \cdot I) == 1$.

```

>>> unbranched_argument(exp(5*I*pi))
pi
>>> unbranched_argument(exp_polar(5*I*pi))
5*pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0

```

See also:

[*diofant.functions.elementary.exponential.exp_polar*](#) (page 295)

[***diofant.functions.elementary.complexes.polar_lift***](#) (page 278)

Lift argument to the Riemann surface of the logarithm

[*diofant.functions.elementary.complexes.principal_branch*](#) (page 279)

principal_branch

class diofant.functions.elementary.complexes.**principal_branch**(*x, period*)

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

This is a function of two arguments. The first argument is a polar number z , and the second one a positive real number of infinity, p . The result is “ $z \bmod \exp_polar(I \cdot p)$ ”.

```

>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)

```

See also:

[*diofant.functions.elementary.exponential.exp_polar*](#) (page 295)

[***diofant.functions.elementary.complexes.polar_lift***](#) (page 278)

Lift argument to the Riemann surface of the logarithm

[*diofant.functions.elementary.complexes.periodic_argument*](#) (page 279)

transpose

class `diofant.functions.elementary.complexes.transpose(arg)`
Linear map transposition.

4.6.3 `diofant.functions.elementary.trigonometric`

4.6.4 Trigonometric Functions

sin

class `diofant.functions.elementary.trigonometric.sin(arg)`
The sine function.
Returns the sine of x (measured in radians).

Notes

This function will evaluate automatically in the case x/π is some rational number. For example, if x is a multiple of π , $\pi/2$, $\pi/3$, $\pi/4$ and $\pi/6$.

Examples

```
>>> sin(x**2).diff(x)
2*x*cos(x**2)
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
1/2
>>> sin(pi/12)
-sqrt(2)/4 + sqrt(6)/4
```

See also:

`diofant.functions.elementary.trigonometric.csc` (page 284), `diofant.functions.elementary.trigonometric.cos` (page 281), `diofant.functions.elementary.trigonometric.sec` (page 283), `diofant.functions.elementary.trigonometric.tan` (page 282), `diofant.functions.elementary.trigonometric.cot` (page 283), `diofant.functions.elementary.trigonometric.asin` (page 285), `diofant.functions.elementary.trigonometric.acsc` (page 288), `diofant.functions.elementary.trigonometric.acos` (page 286), `diofant.functions.elementary.trigonometric.asec` (page 286), `diofant.functions.elementary.trigonometric.atan` (page 287), `diofant.functions.elementary.trigonometric.acot` (page 288), `diofant.functions.elementary.trigonometric.atan2` (page 289)

References

- https://en.wikipedia.org/wiki/Trigonometric_functions
- <https://dlmf.nist.gov/4.14>
- <http://functions.wolfram.com/ElementaryFunctions/Sin>
- <https://mathworld.wolfram.com/TrigonometryAngles.html>

cos

class diofant.functions.elementary.trigonometric.cos(*arg*)

The cosine function.

Returns the cosine of *x* (measured in radians).

Notes

See [sin\(\)](#) (page 280) for notes about automatic evaluation.

Examples

```
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
>>> cos(pi/12)
sqrt(2)/4 + sqrt(6)/4
```

See also:

[diofant.functions.elementary.trigonometric.sin](#) (page 280), [diofant.functions.elementary.trigonometric.csc](#) (page 284), [diofant.functions.elementary.trigonometric.sec](#) (page 283), [diofant.functions.elementary.trigonometric.tan](#) (page 282), [diofant.functions.elementary.trigonometric.cot](#) (page 283), [diofant.functions.elementary.trigonometric.asin](#) (page 285), [diofant.functions.elementary.trigonometric.acsc](#) (page 288), [diofant.functions.elementary.trigonometric.acos](#) (page 286), [diofant.functions.elementary.trigonometric.asec](#) (page 286), [diofant.functions.elementary.trigonometric.atan](#) (page 287), [diofant.functions.elementary.trigonometric.acot](#) (page 288), [diofant.functions.elementary.trigonometric.atan2](#) (page 289)

References

- https://en.wikipedia.org/wiki/Trigonometric_functions
- <https://dlmf.nist.gov/4.14>
- <http://functions.wolfram.com/ElementaryFunctions/Cos>

tan

class `diofant.functions.elementary.trigonometric.tan(arg)`

The tangent function.

Returns the tangent of x (measured in radians).

Notes

See `sin()` (page 280) for notes about automatic evaluation.

Examples

```
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(pi/8).expand()
-1 + sqrt(2)
```

See also:

`diofant.functions.elementary.trigonometric.sin` (page 280), `diofant.functions.elementary.trigonometric.csc` (page 284), `diofant.functions.elementary.trigonometric.cos` (page 281), `diofant.functions.elementary.trigonometric.sec` (page 283), `diofant.functions.elementary.trigonometric.cot` (page 283), `diofant.functions.elementary.trigonometric.asin` (page 285), `diofant.functions.elementary.trigonometric.acsc` (page 288), `diofant.functions.elementary.trigonometric.acos` (page 286), `diofant.functions.elementary.trigonometric.asec` (page 286), `diofant.functions.elementary.trigonometric.atan` (page 287), `diofant.functions.elementary.trigonometric.acot` (page 288), `diofant.functions.elementary.trigonometric.atan2` (page 289)

References

- https://en.wikipedia.org/wiki/Trigonometric_functions
- <https://dlmf.nist.gov/4.14>
- <http://functions.wolfram.com/ElementaryFunctions/Tan>

inverse(`argindex=1`)

Returns the inverse of this function.

cot

class diofant.functions.elementary.trigonometric.cot(*arg*)

The cotangent function.

Returns the cotangent of *x* (measured in radians).

Notes

See [*sin\(\)*](#) (page 280) for notes about automatic evaluation.

Examples

```
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
```

See also:

[*diofant.functions.elementary.trigonometric.sin*](#) (page 280), [*diofant.functions.elementary.trigonometric.csc*](#) (page 284), [*diofant.functions.elementary.trigonometric.cos*](#) (page 281), [*diofant.functions.elementary.trigonometric.sec*](#) (page 283), [*diofant.functions.elementary.trigonometric.tan*](#) (page 282), [*diofant.functions.elementary.trigonometric.asin*](#) (page 285), [*diofant.functions.elementary.trigonometric.acsc*](#) (page 288), [*diofant.functions.elementary.trigonometric.acos*](#) (page 286), [*diofant.functions.elementary.trigonometric.asec*](#) (page 286), [*diofant.functions.elementary.trigonometric.atan*](#) (page 287), [*diofant.functions.elementary.trigonometric.acot*](#) (page 288), [*diofant.functions.elementary.trigonometric.atan2*](#) (page 289)

References

- https://en.wikipedia.org/wiki/Trigonometric_functions
- <https://dlmf.nist.gov/4.14>
- <http://functions.wolfram.com/ElementaryFunctions/Cot>

inverse(*argindex*=1)

Returns the inverse of this function.

sec

class diofant.functions.elementary.trigonometric.sec(*arg*)

The secant function.

Returns the secant of *x* (measured in radians).

Notes

See `sin()` (page 280) for notes about automatic evaluation.

Examples

```
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
```

See also:

`diofant.functions.elementary.trigonometric.sin` (page 280), `diofant.functions.elementary.trigonometric.csc` (page 284), `diofant.functions.elementary.trigonometric.cos` (page 281), `diofant.functions.elementary.trigonometric.tan` (page 282), `diofant.functions.elementary.trigonometric.cot` (page 283), `diofant.functions.elementary.trigonometric.asin` (page 285), `diofant.functions.elementary.trigonometric.acsc` (page 288), `diofant.functions.elementary.trigonometric.acos` (page 286), `diofant.functions.elementary.trigonometric.asec` (page 286), `diofant.functions.elementary.trigonometric.atan` (page 287), `diofant.functions.elementary.trigonometric.acot` (page 288), `diofant.functions.elementary.trigonometric.atan2` (page 289)

References

- https://en.wikipedia.org/wiki/Trigonometric_functions
- <https://dlmf.nist.gov/4.14>
- <http://functions.wolfram.com/ElementaryFunctions/Sec>

csc

class `diofant.functions.elementary.trigonometric.csc(arg)`

The cosecant function.

Returns the cosecant of x (measured in radians).

Notes

See `sin()` (page 280) for notes about automatic evaluation.

Examples

```
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
```


See also:

`diofant.functions.elementary.trigonometric.sin` (page 280), `diofant.functions.elementary.trigonometric.cos` (page 281), `diofant.functions.elementary.trigonometric.sec` (page 283), `diofant.functions.elementary.trigonometric.tan` (page 282), `diofant.functions.elementary.trigonometric.cot` (page 283), `diofant.functions.elementary.trigonometric.asin` (page 285), `diofant.functions.elementary.trigonometric.acsc` (page 288), `diofant.functions.elementary.trigonometric.acos` (page 286), `diofant.functions.elementary.trigonometric.asec` (page 286), `diofant.functions.elementary.trigonometric.atan` (page 287), `diofant.functions.elementary.trigonometric.acot` (page 288), `diofant.functions.elementary.trigonometric.atan2` (page 289)

References

- https://en.wikipedia.org/wiki/Trigonometric_functions
- <https://dlmf.nist.gov/4.14>
- <http://functions.wolfram.com/ElementaryFunctions/Csc>

4.6.5 Trigonometric Inverses

asin

class `diofant.functions.elementary.trigonometric.asin(arg)`

The inverse sine function.

Examples

```
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2
>>> asin(oo)
-oo*I
```

See also:

`sin` (page 280), `csc` (page 284), `acsc` (page 288)

References

- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- <https://dlmf.nist.gov/4.23>
- <http://functions.wolfram.com/ElementaryFunctions/ArcSin>

inverse(*argindex*=1)

Returns the inverse of this function.

acos

class diofant.functions.elementary.trigonometric.acos(*arg*)

The inverse cosine function.

Examples

```
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I
```

See also:

cos (page 281), *sec* (page 283), *asec* (page 286)

References

- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- <https://dlmf.nist.gov/4.23>
- <http://functions.wolfram.com/ElementaryFunctions/ArcCos>

inverse(*argindex*=1)

Returns the inverse of this function.

asec

class diofant.functions.elementary.trigonometric.asec(*arg*)

The inverse secant function.

Examples

```
>>> asec(1)
0
>>> asec(-1)
pi
```

See also:

sec (page 283), *cos* (page 281), *acos* (page 286)

References

- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- <https://dlmf.nist.gov/4.23>
- <http://functions.wolfram.com/ElementaryFunctions/ArcSec>

inverse(*argindex*=1)

Returns the inverse of this function.

atan

class `diofant.functions.elementary.trigonometric.atan`(*arg*)

The inverse tangent function.

Returns the arc tangent of *x* (measured in radians).

Notes

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

Examples

```
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2
```

See also:

`diofant.functions.elementary.trigonometric.sin` (page 280), `diofant.functions.elementary.trigonometric.csc` (page 284), `diofant.functions.elementary.trigonometric.cos` (page 281), `diofant.functions.elementary.trigonometric.sec` (page 283), `diofant.functions.elementary.trigonometric.tan` (page 282), `diofant.functions.elementary.trigonometric.cot` (page 283), `diofant.functions.elementary.trigonometric.asin` (page 285), `diofant.functions.elementary.trigonometric.acsc` (page 288), `diofant.functions.elementary.trigonometric.acos` (page 286), `diofant.functions.elementary.trigonometric.asec` (page 286), `diofant.functions.elementary.trigonometric.acot` (page 288), `diofant.functions.elementary.trigonometric.atan2` (page 289)

References

- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- <https://dlmf.nist.gov/4.23>
- <http://functions.wolfram.com/ElementaryFunctions/ArcTan>

inverse(*argindex=1*)

Returns the inverse of this function.

acot

class diofant.functions.elementary.trigonometric.**acot**(*arg*)

The inverse cotangent function.

Returns the arc cotangent of x (measured in radians). This function has a branch cut discontinuity in the complex plane running from $-i$ to i .

See also:

diofant.functions.elementary.trigonometric.sin (page 280), *diofant.functions.elementary.trigonometric.csc* (page 284), *diofant.functions.elementary.trigonometric.cos* (page 281), *diofant.functions.elementary.trigonometric.sec* (page 283), *diofant.functions.elementary.trigonometric.tan* (page 282), *diofant.functions.elementary.trigonometric.cot* (page 283), *diofant.functions.elementary.trigonometric.asin* (page 285), *diofant.functions.elementary.trigonometric.acsc* (page 288), *diofant.functions.elementary.trigonometric.acos* (page 286), *diofant.functions.elementary.trigonometric.asec* (page 286), *diofant.functions.elementary.trigonometric.atan* (page 287), *diofant.functions.elementary.trigonometric.atan2* (page 289)

References

- <https://dlmf.nist.gov/4.23>
- <http://functions.wolfram.com/ElementaryFunctions/ArcCot>
- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions

inverse(*argindex=1*)

Returns the inverse of this function.

acsc

class diofant.functions.elementary.trigonometric.**acsc**(*arg*)

The inverse cosecant function.

Examples

```
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
```

See also:

[csc](#) (page 284), [sin](#) (page 280), [asin](#) (page 285)

References

- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- <https://dlmf.nist.gov/4.23>
- <http://functions.wolfram.com/ElementaryFunctions/ArcCsc>

inverse(*argindex*=1)

Returns the inverse of this function.

atan2

class diofant.functions.elementary.trigonometric.**atan2**(*y*, *x*)

The function `atan2(y, x)` computes $\operatorname{atan}(y/x)$ taking two arguments *y* and *x*. Signs of both *y* and *x* are considered to determine the appropriate quadrant of $\operatorname{atan}(y/x)$. The range is $(-\pi, \pi]$. The complete definition reads as follows:

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Attention: Note the role reversal of both arguments. The *y*-coordinate is the first argument and the *x*-coordinate the second.

Examples

Going counter-clock wise around the origin we find the following angles:

```
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
```

(continues on next page)

(continued from previous page)

```
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary `atan` function for the point $(x, y) = (-1, 1)$

```
>>> atan(Integer(1) / -1)
-pi/4
>>> atan2(1, -1)
3*pi/4
```

where only the `atan2` function returns what we expect. We can differentiate the function with respect to both arguments:

```
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)
```

```
>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the `atan2` function in terms of complex logarithms:

```
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of (*atan*):

```
>>> atan2(y, x).rewrite(atan)
2*atan(y/(x + sqrt(x**2 + y**2)))
```

but note that this form is undefined on the negative real axis.

See also:

diofant.functions.elementary.trigonometric.sin (page 280), *diofant.functions.elementary.trigonometric.csc* (page 284), *diofant.functions.elementary.trigonometric.cos* (page 281), *diofant.functions.elementary.trigonometric.sec* (page 283), *diofant.functions.elementary.trigonometric.tan* (page 282), *diofant.functions.elementary.trigonometric.cot* (page 283), *diofant.functions.elementary.trigonometric.asin* (page 285), *diofant.functions.elementary.trigonometric.acsc* (page 288), *diofant.functions.elementary.trigonometric.acos* (page 286), *diofant.functions.elementary.trigonometric.asec* (page 286), *diofant.functions.elementary.trigonometric.atan* (page 287), *diofant.functions.elementary.trigonometric.acot* (page 288)

References

- https://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- <https://en.wikipedia.org/wiki/Atan2>
- <http://functions.wolfram.com/ElementaryFunctions/ArcTan2>

4.6.6 diofant.functions.elementary.hyperbolic

4.6.7 Hyperbolic Functions

HyperbolicFunction

class diofant.functions.elementary.hyperbolic.**HyperbolicFunction**(*args)

Base class for hyperbolic functions.

See also:

diofant.functions.elementary.hyperbolic.sinh (page 291), *diofant.functions.elementary.hyperbolic.cosh* (page 291), *diofant.functions.elementary.hyperbolic.tanh* (page 292), *diofant.functions.elementary.hyperbolic.coth* (page 292)

sinh

class diofant.functions.elementary.hyperbolic.**sinh**(arg)

The hyperbolic sine function, $\frac{e^x - e^{-x}}{2}$.

- $\sinh(x)$ -> Returns the hyperbolic sine of x

See also:

diofant.functions.elementary.hyperbolic.cosh (page 291), *diofant.functions.elementary.hyperbolic.tanh* (page 292), *diofant.functions.elementary.hyperbolic.asinh* (page 293)

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(n, x, *previous_terms)

Returns the next term in the Taylor series expansion.

cosh

class diofant.functions.elementary.hyperbolic.**cosh**(arg)

The hyperbolic cosine function, $\frac{e^x + e^{-x}}{2}$.

- $\cosh(x)$ -> Returns the hyperbolic cosine of x

See also:

diofant.functions.elementary.hyperbolic.sinh (page 291), *diofant.functions.elementary.hyperbolic.tanh* (page 292), *diofant.functions.elementary.hyperbolic.acosh* (page 293)

tanh

class diofant.functions.elementary.hyperbolic.tanh(*arg*)

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

- tanh(x) -> Returns the hyperbolic tangent of x

See also:

[diofant.functions.elementary.hyperbolic.sinh](#) (page 291), [diofant.functions.elementary.hyperbolic.cosh](#) (page 291), [diofant.functions.elementary.hyperbolic.atanh](#) (page 293)

inverse(*argindex*=1)

Returns the inverse of this function.

coth

class diofant.functions.elementary.hyperbolic.coth(*arg*)

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.

sech

class diofant.functions.elementary.hyperbolic.sech(*arg*)

The hyperbolic secant function, $\frac{2}{e^x + e^{-x}}$

- sech(x) -> Returns the hyperbolic secant of x

See also:

[diofant.functions.elementary.hyperbolic.sinh](#) (page 291), [diofant.functions.elementary.hyperbolic.cosh](#) (page 291), [diofant.functions.elementary.hyperbolic.tanh](#) (page 292), [diofant.functions.elementary.hyperbolic.coth](#) (page 292), [diofant.functions.elementary.hyperbolic.csch](#) (page 292), [diofant.functions.elementary.hyperbolic.asinh](#) (page 293), [diofant.functions.elementary.hyperbolic.acosh](#) (page 293)

csch

class diofant.functions.elementary.hyperbolic.csch(*arg*)

The hyperbolic cosecant function, $\frac{2}{e^x - e^{-x}}$

- csch(x) -> Returns the hyperbolic cosecant of x

See also:

[diofant.functions.elementary.hyperbolic.sinh](#) (page 291), [diofant.functions.elementary.hyperbolic.cosh](#) (page 291), [diofant.functions.elementary.hyperbolic.tanh](#) (page 292), [diofant.functions.elementary.hyperbolic.sech](#)

(page 292), *diofant.functions.elementary.hyperbolic.asinh* (page 293), *diofant.functions.elementary.hyperbolic.acosh* (page 293)

fdiff(*argindex*=1)

Returns the first derivative of this function.

static taylor_term(*n*, *x*, **previous_terms*)

Returns the next term in the Taylor series expansion.

4.6.8 Hyperbolic Inverses

asinh

class *diofant.functions.elementary.hyperbolic.asinh*(*arg*)

The inverse hyperbolic sine function.

See also:

sinh (page 291)

inverse(*argindex*=1)

Returns the inverse of this function.

acosh

class *diofant.functions.elementary.hyperbolic.acosh*(*arg*)

The inverse hyperbolic cosine function.

See also:

cosh (page 291)

inverse(*argindex*=1)

Returns the inverse of this function.

atanh

class *diofant.functions.elementary.hyperbolic.atanh*(*arg*)

The inverse hyperbolic tangent function.

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

See also:

diofant.functions.elementary.hyperbolic.asinh (page 293), *diofant.functions.elementary.hyperbolic.acosh* (page 293), *diofant.functions.elementary.hyperbolic.tanh* (page 292)

inverse(*argindex*=1)

Returns the inverse of this function.

acoth

class diofant.functions.elementary.hyperbolic.**acoth**(*arg*)

The inverse hyperbolic cotangent function.

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

inverse(*argindex*=1)

Returns the inverse of this function.

4.6.9 diofant.functions.elementary.integers

ceiling

class diofant.functions.elementary.integers.**ceiling**(*arg*)

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.

Examples

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

See also:

diofant.functions.elementary.integers.floor (page 294)

References

- “Concrete mathematics” by Graham, pp. 87
- <https://mathworld.wolfram.com/CeilingFunction.html>

floor

class diofant.functions.elementary.integers.**floor**(*arg*)

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

Examples

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

See also:

diofant.functions.elementary.integers.ceiling (page 294)

References

- “Concrete mathematics” by Graham, pp. 87
- <https://mathworld.wolfram.com/FloorFunction.html>

RoundFunction

class `diofant.functions.elementary.integers.RoundFunction(arg)`

The base class for rounding functions.

4.6.10 diofant.functions.elementary.exponential

exp

`diofant.functions.elementary.exponential.exp(arg, **kwargs)`

The exponential function, e^x .

See also:

diofant.functions.elementary.exponential.log (page 296)

exp_polar

class `diofant.functions.elementary.exponential.exp_polar(*args)`

Represent a ‘polar number’ (see g-function Sphinx documentation).

`exp_polar` represents the function $Exp : \mathbb{C} \rightarrow S$, sending the complex number $z = a + bi$ to the polar number $r = \exp(a), \theta = b$. It is one of the main functions to construct polar numbers.

The main difference is that polar numbers don’t “wrap around” at 2π :

```
>>> exp(2*pi*I)
1
>>> exp_polar(2*pi*I)
exp_polar(2*I*pi)
```

apart from that they behave mostly like classical complex numbers:

```
>>> exp_polar(2)*exp_polar(3)
exp_polar(5)
```

See also:

diofant.simplify.powsimp.powsimp (page 596), *diofant.functions.elementary.complexes.polar_lift* (page 278), *diofant.functions.elementary.complexes.periodic_argument* (page 279), *diofant.functions.elementary.complexes.principal_branch* (page 279)

property `exp`

Returns the exponent of the function.

LambertW

class `diofant.functions.elementary.exponential.LambertW(x, k=None)`

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 $z = w \exp(w)$.

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 $z = 0$.

Examples

```
>>> LambertW(1.2)
0.635564016364870
>>> LambertW(1.2, -1).evalf()
-1.34747534407696 - 4.41624341514535*I
>>> LambertW(-1).is_real
False
```

References

- https://en.wikipedia.org/wiki/Lambert_W_function

fdiff(*argindex*=1)

Return the first derivative of this function.

log

class `diofant.functions.elementary.exponential.log(arg, base=None)`

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:

diofant.functions.elementary.exponential.exp (page 295)

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

Returns this function as a complex coordinate.

Examples

```
>>> 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 $\log(x)$.

4.6.11 diofant.functions.elementary.piecewise

ExprCondPair

class diofant.functions.elementary.piecewise.**ExprCondPair**(*expr, cond*)

Represents an expression, condition pair.

property cond

Returns the condition of this pair.

property expr

Returns the expression of this pair.

Piecewise

class diofant.functions.elementary.piecewise.**Piecewise**(**args*)

Represents a piecewise function.

Usage:

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

- Each argument is a 2-tuple defining an expression and condition
- The conds are evaluated in turn returning the first that is True. If any of the evaluated conds are not determined explicitly False, e.g. $x < 1$, the function is returned in symbolic form.
- Pairs where the cond is explicitly False, will be removed.
- The last condition is explicitly True. It's value is a default value for the function (nan if not specified otherwise).

Examples

```
>>> 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)
```

See also:

[*diofant.functions.elementary.piecewise.piecewise_fold*](#) (page 298)

doit(***hints*)

Evaluate this piecewise function.

`diofant.functions.elementary.piecewise.piecewise_fold(expr)`

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

Examples

```
>>> p = Piecewise((x, x < 1), (1, True))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, true))
```

See also:

[*diofant.functions.elementary.piecewise.Piecewise*](#) (page 297)

4.6.12 diofant.functions.elementary.miscellaneous

IdentityFunction

class `diofant.functions.elementary.miscellaneous.IdentityFunction`(**args*,
***kwargs*)

The identity function

Examples

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

Min

class diofant.functions.elementary.miscellaneous.Min(*args)

Return, if possible, the minimum value of the list.

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

Examples

```
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Min(x, -2)
Min(-2, x)
>>> 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(-7, n)
```

See also:

[*diofant.functions.elementary.miscellaneous.Max*](#) (page 299)
find maximum values

Max

class diofant.functions.elementary.miscellaneous.Max(*args)

Return, if possible, the maximum value of the list.

When number of arguments is equal one, then return this argument.

When number of arguments is equal two, then return, if possible, the value from (a, b) that is \geq the other.

In common case, when the length of list greater than 2, the task is more complicated. Return only the arguments, which are greater than others, if it is possible to determine directional relation.

If is not possible to determine such a relation, return a partially evaluated result.

Assumptions are used to make the decision too.

Also, only comparable arguments are permitted.

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

Examples

```
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Max(x, -2)
Max(-2, x)
>>> 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
```

Notes

The task can be considered as searching of supremums in the directed complete partial orders.

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

References

- https://en.wikipedia.org/wiki/Directed_complete_partial_order
- https://en.wikipedia.org/wiki/Lattice_%28order%29

See also:

diofant.functions.elementary.miscellaneous.Min (page 299)
find minimum values

root

`diofant.functions.elementary.miscellaneous.root(arg, n, k=0, **kwargs)`

Returns the k-th n-th root of arg, defaulting to the principle root.

Examples

```
>>> root(x, 2)
sqrt(x)
```

```
>>> root(x, 3)
root(x, 3)
```

```
>>> root(x, n)
x**(1/n)
```

```
>>> root(x, -Rational(2, 3))
1/sqrt(x)**3
```

To get the k-th n-th root, specify k:

```
>>> root(-2, 3, 2)
-root(-1, 3)**2*root(2, 3)
```

To get all n n-th roots you can use the `RootOf` function. The following examples show the roots of unity for n equal 2, 3 and 4:

```
>>> [RootOf(x**2 - 1, i) for i in range(2)]
[-1, 1]
```

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

```
>>> [RootOf(x**4 - 1, i) for i in range(4)]
[-1, 1, -I, I]
```

Diofant, 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*root(-1, 3)
```

The `real_root` function can be used to either make the principle result real (or simply to return the real root directly):

```
>>> real_root(_)
-2
>>> real_root(-32, 5)
-2
```

Alternatively, the n//2-th n-th root of a negative number can be computed with `root`:

```
>>> root(-32, 5, 5//2)
-2
```

See also:

[`diofant.polys.rootoftools.RootOf`](#) (page 540), [`diofant.core.power.integer_nthroot`](#) (page 101), [`diofant.functions.elementary.miscellaneous.sqrt`](#) (page 302), [`diofant.functions.elementary.miscellaneous.real_root`](#) (page 302)

References

- https://en.wikipedia.org/wiki/Square_root
- https://en.wikipedia.org/wiki/Real_root
- https://en.wikipedia.org/wiki/Root_of_unity
- https://en.wikipedia.org/wiki/Principal_value
- <https://mathworld.wolfram.com/CubeRoot.html>

real_root

`diofant.functions.elementary.miscellaneous.real_root(arg, n=None)`

Return the real n th-root of arg if possible. If n is omitted then all instances of $(-n)^{1/odd}$ will be changed to $-n^{1/odd}$; this will only create a real root of a principle root – the presence of other factors may cause the result to not be real.

Examples

```
>>> real_root(-8, 3)
-2
>>> root(-8, 3)
2*root(-1, 3)
>>> real_root(_)
-2
```

If one creates a non-principle root and applies `real_root`, the result will not be real (so use with caution):

```
>>> root(-8, 3, 2)
-2*root(-1, 3)**2
>>> real_root(_)
-2*root(-1, 3)**2
```

See also:

[`diofant.polys.rootoftools.RootOf`](#) (page 540), [`diofant.core.power.integer_nthroot`](#) (page 101), [`diofant.functions.elementary.miscellaneous.root`](#) (page 301), [`diofant.functions.elementary.miscellaneous.sqrt`](#) (page 302)

sqrt

`diofant.functions.elementary.miscellaneous.sqrt(arg, **kwargs)`

The square root function

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

Examples

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

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

Note that $\text{sqrt}(x^{**2})$ 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 $x == -1$:

```
>>> Eq(sqrt(x**2), x).subs({x: -1})
false
```

This is because sqrt computes the principal square root, so the square may put the argument in a different branch. This identity does hold if x is positive:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the `powdenest()` function with the `force` option set to `True`:

```
>>> 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:

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

See also:

diofant.polys.rootoftools.RootOf (page 540), *diofant.functions.elementary.miscellaneous.root* (page 301), *diofant.functions.elementary.miscellaneous.real_root* (page 302)

References

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

4.6.13 Combinatorial

This module implements various combinatorial functions.

bell

class diofant.functions.combinatorial.numbers.**bell**(*n*, *k_sym*=None, *symbols*=None)

Bell numbers / Bell polynomials

The Bell numbers satisfy $B_0 = 1$ and

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$

They are also given by:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

The Bell polynomials are given by $B_0(x) = 1$ and

$$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$

The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots, x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_3+\dots=k \\ j_1+2j_2+3j_3+\dots=n}} \frac{n!}{j_1!j_2!\dots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \dots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- `bell(n)` gives the n^{th} Bell number, B_n .
- `bell(n, x)` gives the n^{th} Bell polynomial, $B_n(x)$.
- `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$.

Notes

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

Examples

```
>>> [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
```

References

- https://en.wikipedia.org/wiki/Bell_number
- <https://mathworld.wolfram.com/BellNumber.html>
- <https://mathworld.wolfram.com/BellPolynomial.html>

See also:

`diofant.functions.combinatorial.numbers.bernoulli` (page 305), `diofant.functions.combinatorial.numbers.catalan` (page 307), `diofant.functions.combinatorial.numbers.euler` (page 309), `diofant.functions.combinatorial.numbers.fibonacci` (page 312), `diofant.functions.combinatorial.numbers.harmonic` (page 313), `diofant.functions.combinatorial.numbers.lucas` (page 315)

bernoulli

class `diofant.functions.combinatorial.numbers.bernoulli(n, sym=None)`

Bernoulli numbers / Bernoulli polynomials

The Bernoulli numbers are a sequence of rational numbers defined by $B_0 = 1$ and the recursive relation ($n > 0$):

$$0 = \sum_{k=0}^n \binom{n+1}{k} B_k.$$

They are also commonly defined by their exponential generating function, which is $x/(\exp(x) - 1)$. For odd indices > 1 , the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \sum_{k=0}^n \binom{n}{k} B_k x^{n-k}.$$

Bernoulli numbers and Bernoulli polynomials are related as $B_n(0) = B_n$.

We compute Bernoulli numbers using Ramanujan's formula:

$$B_n = (A(n) - S(n)) / \binom{n+3}{n}$$

where $A(n) = (n+3)/3$ when $n = 0$ or $2 \pmod{6}$, $A(n) = -(n+3)/6$ when $n = 4 \pmod{6}$, and:

$$S(n) = \sum_{k=1}^{\lfloor n/6 \rfloor} \binom{n+3}{n-6k} B_{n-6k}$$

This formula is similar to the sum given in the definition, but cuts 2/3 of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the n th Bernoulli number, B_n
- `bernoulli(n, x)` gives the n th Bernoulli polynomial in x , $B_n(x)$

Examples

```
>>> [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
```

References

- https://en.wikipedia.org/wiki/Bernoulli_number
- https://en.wikipedia.org/wiki/Bernoulli_polynomial
- <https://mathworld.wolfram.com/BernoulliNumber.html>
- <https://mathworld.wolfram.com/BernoulliPolynomial.html>

See also:

diofant.functions.combinatorial.numbers.bell (page 304), *diofant.functions.combinatorial.numbers.catalan* (page 307), *diofant.functions.combinatorial.numbers.euler* (page 309), *diofant.functions.combinatorial.numbers.fibonacci* (page 312), *diofant.functions.combinatorial.numbers.harmonic* (page 313), *diofant.functions.combinatorial.numbers.lucas* (page 315)

binomial

class `diofant.functions.combinatorial.factorials.binomial(n, k)`

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

$$C(n, k) = n! / (k!(n-k)!) \text{ or } C(n, k) = \text{ff}(n, k) / k!$$

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

```
>>> n = Symbol('n', integer=True, positive=True)
```

```
>>> binomial(15, 8)
6435
```

```
>>> binomial(n, -1)
0
```

Rows of Pascal's triangle can be generated with the binomial function:

```
>>> for N in range(8):
    [binomial(N, i) for i in range(N + 1)]
[[1]]
[[1, 1]]
[[1, 2, 1]]
[[1, 3, 3, 1]]
[[1, 4, 6, 4, 1]]
[[1, 5, 10, 10, 5, 1]]
[[1, 6, 15, 20, 15, 6, 1]]
[[1, 7, 21, 35, 35, 21, 7, 1]]
```

As can a given diagonal, e.g. the 4th diagonal:

```
>>> N = -4
>>> [binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]
```

```
>>> binomial(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 diofant.functions.combinatorial.numbers.**catalan**(n)

Catalan numbers

The n-th catalan number is given by:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

- `catalan(n)` gives the n-th Catalan number, C_n

Examples

```
>>> [catalan(i) for i in range(1, 10)]  
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> catalan(n)  
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```
>>> catalan(n).rewrite(binomial)  
binomial(2*n, n)/(n + 1)
```

```
>>> catalan(n).rewrite(gamma)  
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))
```

```
>>> catalan(n).rewrite(hyper)  
hyper((-n + 1, -n), (2,), 1)
```

For some non-integer values of n we can get closed form expressions by rewriting in terms of gamma functions:

```
>>> catalan(Rational(1, 2)).rewrite(gamma)  
8/(3*pi)
```

We can differentiate the Catalan numbers $C(n)$ interpreted as a continuous real function in n :

```
>>> diff(catalan(n), n)  
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```

As a more advanced example consider the following ratio between consecutive numbers:

```
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))  
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```
>>> catalan(I).rewrite(gamma)  
4*I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```
>>> catalan(I).evalf(20)  
0.39764993382373624267 - 0.020884341620842555705*I
```

References

- https://en.wikipedia.org/wiki/Catalan_number
- <https://mathworld.wolfram.com/CatalanNumber.html>
- <http://functions.wolfram.com/GammaBetaErf/CatalanNumber/>
- <http://geometer.org/mathcircles/catalan.pdf>

See also:

diofant.functions.combinatorial.numbers.bell (page 304), *diofant.functions.combinatorial.numbers.bernoulli* (page 305), *diofant.functions.combinatorial*.

numbers.euler (page 309), *diofant.functions.combinatorial.numbers.fibonacci* (page 312), *diofant.functions.combinatorial.numbers.harmonic* (page 313), *diofant.functions.combinatorial.numbers.lucas* (page 315), *diofant.functions.combinatorial.factorials.binomial* (page 306)

euler

class `diofant.functions.combinatorial.numbers.euler(m)`

Euler numbers

The euler numbers are given by:

$$E_{2n} = I \int_0^{\infty} \prod_{k=1}^{\infty} \left(1 - \frac{x^{2k}}{2^k} \right)^{(-1)^k} dx$$

$$E_{2n+1} = 0$$

- `euler(n)` gives the n-th Euler number, E_n

Examples

```
>>> from diofant.functions import euler
```

```
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> euler(n+2*n)
euler(3*n)
```

References

- https://en.wikipedia.org/wiki/Euler_numbers
- <https://mathworld.wolfram.com/EulerNumber.html>
- https://en.wikipedia.org/wiki/Alternating_permutation
- <https://mathworld.wolfram.com/AlternatingPermutation.html>

See also:

diofant.functions.combinatorial.numbers.bell (page 304), *diofant.functions.combinatorial.numbers.bernoulli* (page 305), *diofant.functions.combinatorial.numbers.fibonacci* (page 312), *diofant.functions.combinatorial.numbers.harmonic* (page 313), *diofant.functions.combinatorial.numbers.lucas* (page 315)

factorial

class diofant.functions.combinatorial.factorials.**factorial**(*n*)

Implementation of factorial function over nonnegative integers.

By convention (consistent with the gamma function and the binomial coefficients), factorial of a negative integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in which n objects can be permuted. It also arises in calculus, probability, number theory, etc.

There is strict relation of factorial with gamma function. In fact $n! = \text{gamma}(n+1)$ for nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simplification.

Computation of the factorial is done using two algorithms. For small arguments 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'.

Examples

```
>>> factorial(0)
1
```

```
>>> factorial(7)
5040
```

```
>>> factorial(-2)
zoo
```

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

```
>>> factorial(2*n)
factorial(2*n)
```

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

See also:

[*diofant.functions.combinatorial.factorials.factorial2*](#) (page 311), [*diofant.functions.combinatorial.factorials.RisingFactorial*](#) (page 316), [*diofant.functions.combinatorial.factorials.FallingFactorial*](#) (page 312)

subfactorial

class diofant.functions.combinatorial.factorials.**subfactorial**(*arg*)

The subfactorial counts the derangements of n items and is defined for non-negative integers as:

$$!n = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{for } n = 1 \\ (n - 1) * (!n - 1 + !n - 2) & \text{for } n > 1 \end{cases}$$

It can also be written as `int(round(n!/exp(1)))` but the recursive definition with caching is implemented for this function.

An interesting analytic expression is the following

$$!x = \Gamma(x + 1, -1)/e$$

which is valid for non-negative integers x . The above formula is not very useful in case of non-integers. $\Gamma(x + 1, -1)$ is single-valued only for integral arguments x , elsewhere on the positive real axis it has an infinite number of branches none of which are real.

References

- <https://en.wikipedia.org/wiki/Subfactorial>
- <https://mathworld.wolfram.com/Subfactorial.html>

Examples

```
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

See also:

diofant.functions.combinatorial.factorials.factorial (page 310), *diofant.functions.special.gamma_functions.uppergamma* (page 325)

factorial2 / double factorial

class `diofant.functions.combinatorial.factorials.factorial2(n)`

The double factorial $n!!$, not to be confused with $(n!)!$

The double factorial is defined for nonnegative integers and for odd negative integers as:

$$n!! = \begin{cases} n*(n-2)*(n-4)*\dots*1 & \text{for } n \text{ positive odd} \\ n*(n-2)*(n-4)*\dots*2 & \text{for } n \text{ positive even} \\ 1 & \text{for } n = 0 \\ (n+2)!! / (n+2) & \text{for } n \text{ negative odd} \end{cases}$$

References

- https://en.wikipedia.org/wiki/Double_factorial

Examples

```
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
>>> factorial2(-5)
1/3
```

See also:

diofant.functions.combinatorial.factorials.factorial (page 310), *diofant.functions.combinatorial.factorials.RisingFactorial* (page 316), *diofant.functions.combinatorial.factorials.FallingFactorial* (page 312)

FallingFactorial

class `diofant.functions.combinatorial.factorials.FallingFactorial(x, k)`

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 <https://mathworld.wolfram.com/FallingFactorial.html> page.

```
>>> ff(x, 0)
1
```

```
>>> ff(5, 5)
120
```

```
>>> ff(x, 5) == x*(x-1)*(x-2)*(x-3)*(x-4)
True
```

See also:

diofant.functions.combinatorial.factorials.factorial (page 310), *diofant.functions.combinatorial.factorials.factorial2* (page 311), *diofant.functions.combinatorial.factorials.RisingFactorial* (page 316)

fibonacci

class `diofant.functions.combinatorial.numbers.fibonacci(n, sym=None)`

Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms $F_0 = 0$, $F_1 = 1$ and the two-term recurrence relation $F_n = F_{n-1} + F_{n-2}$. This definition extended to arbitrary real and complex arguments using the formula

$$F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

The Fibonacci polynomials are defined by $F_1(x) = 1$, $F_2(x) = x$, and $F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$ for $n > 2$. For all positive integers n , $F_n(1) = F_n$.

- `fibonacci(n)` gives the n th Fibonacci number, F_n
- `fibonacci(n, x)` gives the n th Fibonacci polynomial in x , $F_n(x)$

Examples

```
>>> [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
```

References

- https://en.wikipedia.org/wiki/Fibonacci_number
- <https://mathworld.wolfram.com/FibonacciNumber.html>

See also:

diofant.functions.combinatorial.numbers.bell (page 304), *diofant.functions.combinatorial.numbers.bernoulli* (page 305), *diofant.functions.combinatorial.numbers.catalan* (page 307), *diofant.functions.combinatorial.numbers.euler* (page 309), *diofant.functions.combinatorial.numbers.harmonic* (page 313), *diofant.functions.combinatorial.numbers.lucas* (page 315)

harmonic

class `diofant.functions.combinatorial.numbers.harmonic(n, m=None)`

Harmonic numbers

The n th harmonic number is given by $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$.

More generally:

$$H_{n,m} = \sum_{k=1}^n \frac{1}{k^m}$$

As $n \rightarrow \infty$, $H_{n,m} \rightarrow \zeta(m)$, the Riemann zeta function.

- `harmonic(n)` gives the n th harmonic number, H_n
- `harmonic(n, m)` gives the n th generalized harmonic number of order m , $H_{n,m}$, where `harmonic(n) == harmonic(n, 1)`

Examples

```
>>> [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
```

```
>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))
```

We can evaluate harmonic numbers for all integral and positive rational arguments:

```
>>> harmonic(8)
761/280
>>> harmonic(11)
83711/27720
```

```
>>> H = harmonic(Rational(1, 3))
>>> H
harmonic(1/3)
>>> He = expand_func(H)
>>> He
-log(6) - sqrt(3)*pi/6 + 2*Sum(log(sin(pi*k/3))*cos(2*pi*_k/3), (_k, 1, 1))
+ 3*Sum(1/(3*_k + 1), (_k, 0, 0))
>>> He.doit()
-log(6) - sqrt(3)*pi/6 - log(sqrt(3)/2) + 3
>>> H = harmonic(Rational(25, 7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(pi/7)**(-2*cos(pi/7))*sin(2*pi/7)**(2*cos(16*pi/7))*cos(pi/14)**(-
2*sin(pi/14))/14)
+ pi*tan(pi/14)/2 + 30247/9900
>>> He.evalf(40)
1.983697455232980674869851942390639915940
>>> harmonic(Rational(25, 7)).evalf(40)
1.983697455232980674869851942390639915940
```

We can rewrite harmonic numbers in terms of polygamma functions:

```
>>> 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:

```
>>> 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:

```
>>> limit(harmonic(n), n, oo)
oo
```

```
>>> limit(harmonic(n, 2), n, oo)
pi**2/6
```

```
>>> limit(harmonic(n, 3), n, oo)
-polygamma(2, 1)/2
```

However we can not compute the general relation yet:

```
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals $\zeta(m)$ for $m > 1$.

References

- https://en.wikipedia.org/wiki/Harmonic_number
- <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/>
- <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/>

See also:

diofant.functions.combinatorial.numbers.bell (page 304), *diofant.functions.combinatorial.numbers.bernoulli* (page 305), *diofant.functions.combinatorial.numbers.catalan* (page 307), *diofant.functions.combinatorial.numbers.euler* (page 309), *diofant.functions.combinatorial.numbers.fibonacci* (page 312), *diofant.functions.combinatorial.numbers.lucas* (page 315)

lucas

class `diofant.functions.combinatorial.numbers.lucas(n)`

Lucas numbers

Lucas numbers satisfy a recurrence relation similar to that of the Fibonacci sequence, in which each term is the sum of the preceding two. They are generated by choosing the initial values $L_0 = 2$ and $L_1 = 1$.

- `lucas(n)` gives the *n*th Lucas number

Examples

```
>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

References

- https://en.wikipedia.org/wiki/Lucas_number
- <https://mathworld.wolfram.com/LucasNumber.html>

See also:

diofant.functions.combinatorial.numbers.bell (page 304), *diofant.functions.combinatorial.numbers.bernoulli* (page 305), *diofant.functions.combinatorial.numbers.catalan* (page 307), *diofant.functions.combinatorial.numbers.euler* (page 309), *diofant.functions.combinatorial.numbers.fibonacci* (page 312), *diofant.functions.combinatorial.numbers.harmonic* (page 313)

RisingFactorial

class diofant.functions.combinatorial.factorials.**RisingFactorial**(*x*, *k*)

Rising factorial (also called Pochhammer symbol) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.

It is defined by:

$$\text{rf}(x, k) = x * (x+1) * \dots * (x + k-1)$$

where 'x' can be arbitrary expression and 'k' is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or visit <https://mathworld.wolfram.com/RisingFactorial.html> page.

Examples

```
>>> rf(x, 0)
1
```

```
>>> rf(1, 5)
120
```

```
>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
True
```

See also:

diofant.functions.combinatorial.factorials.factorial (page 310), *diofant.functions.combinatorial.factorials.factorial2* (page 311), *diofant.functions.combinatorial.factorials.FallingFactorial* (page 312)

stirling

diofant.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 $\text{bell}(n)$. The recurrence relationship for these numbers is:

$$\begin{Bmatrix} 0 \\ 0 \end{Bmatrix} = 1; \quad \begin{Bmatrix} n \\ 0 \end{Bmatrix} = \begin{Bmatrix} 0 \\ k \end{Bmatrix} = 0; \quad \begin{Bmatrix} n+1 \\ k \end{Bmatrix} = j * \begin{Bmatrix} n \\ k \end{Bmatrix} + \begin{Bmatrix} n \\ k-1 \end{Bmatrix}$$

where *j* is::

n for Stirling numbers of the first kind - *n* for signed Stirling numbers of the first kind
k for Stirling numbers of the second kind

The first kind of Stirling number counts the number of permutations of n distinct items that have k cycles; the second kind counts the ways in which n distinct items can be partitioned into k parts. If d is given, the "reduced Stirling number of the second kind" is returned: $S^{\{d\}}(n, k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$. (This counts the ways to partition n consecutive integers into k groups with no pairwise difference less than d . See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword `signed=True`. Using this keyword automatically sets `kind` to 1.

Examples

```
>>> import itertools
>>> from diofant.utilities.iterables import multiset_partitions
```

First kind (unsigned by default):

```
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(itertools.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:

```
>>> def delta(p):
...     if len(p) == 1:
...         return 0
...     return min(abs(i[0] - i[1]) for i in itertools.combinations(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
```

References

- https://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind
- https://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind

See also:

diofant.utilities.iterables.multiset_partitions (page 736)

4.6.14 Special

DiracDelta

class diofant.functions.special.delta_functions.**DiracDelta**(arg, k=Integer(0))

The DiracDelta function and its derivatives.

DiracDelta function has the following properties:

- 1) $\text{diff}(\text{Heaviside}(x), x) = \text{DiracDelta}(x)$
- 2) $\int \text{DiracDelta}(x-a) \cdot f(x) \cdot (x, -\infty, \infty) = f(a)$ and $\int \text{DiracDelta}(x-a) \cdot f(x) \cdot (x, a-e, a+e) = f(a)$
- 3) $\text{DiracDelta}(x) = 0$ for all $x \neq 0$
- 4) $\text{DiracDelta}(g(x)) = \sum_i \text{DiracDelta}(x-x_i) / \text{abs}(g'(x_i))$ Where x_i -s are the roots of g

Derivatives of k-th order of DiracDelta have the following property:

- 5) $\text{DiracDelta}(x, k) = 0$, for all $x \neq 0$

See also:

diofant.functions.special.delta_functions.Heaviside (page 319), *diofant.simplify.simplify.simplify* (page 580), *diofant.functions.special.delta_functions.DiracDelta.is_simple* (page 318), *diofant.functions.special.tensor_functions.KroneckerDelta* (page 388)

References

- <https://mathworld.wolfram.com/DeltaFunction.html>

is_simple(self, x)

Tells whether the argument(args[0]) of DiracDelta is a linear expression in x.

x can be:

- a symbol

Examples

```
>>> 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
```

See also:

diofant.simplify.simplify.simplify (page 580), *diofant.functions.special.delta_functions.DiracDelta* (page 318)

simplify(self, x)

Compute a simplified representation of the function using property number 4.

x can be:

- a symbol

Examples

```
>>> 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
```

See also:

diofant.functions.special.delta_functions.DiracDelta.is_simple
(page 318), *diofant.functions.special.delta_functions.DiracDelta*
(page 318)

Heaviside

class diofant.functions.special.delta_functions.**Heaviside**(arg)

Heaviside step function

$$H(x) = \begin{cases} 0, & x < 0 \\ 1/2, & x = 0 \\ 1, & x > 0 \end{cases}$$

See also:

diofant.functions.special.delta_functions.DiracDelta (page 318)

References

- https://en.wikipedia.org/wiki/Heaviside_step_function

Gamma, Beta and related Functions

class diofant.functions.special.gamma_functions.**gamma**(arg)

The gamma function

$$\Gamma(x) := \int_0^{\infty} t^{x-1} e^{-t} dt.$$

The gamma function implements the function which passes through the values of the factorial function, i.e. $\Gamma(n) = (n-1)!$ when n is an integer. More general, $\Gamma(z)$ is defined in the whole complex plane except at the negative integers where there are simple poles.

Examples

Several special values are known:

```
>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(Rational(3, 2))
sqrt(pi)/2
```

The Gamma function obeys the mirror symmetry:

```
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> gamma(x).series(x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-EulerGamma*pi**2/12 +
↳ polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.288037795340032417959588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

See also:

[lowergamma](#) (page 326)

Lower incomplete gamma function.

[uppergamma](#) (page 325)

Upper incomplete gamma function.

[polygamma](#) (page 322)

Polygamma function.

[loggamma](#) (page 321)

Log Gamma function.

[digamma](#) (page 324)

Digamma function.

[trigamma](#) (page 325)

Trigamma function.

[diofant.functions.special.beta_functions.beta](#) (page 328)

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Gamma_function
- <https://dlmf.nist.gov/5>
- <https://mathworld.wolfram.com/GammaFunction.html>
- <http://functions.wolfram.com/GammaBetaErf/Gamma/>

class diofant.functions.special.gamma_functions.**loggamma**(z)

The loggamma function implements the logarithm of the gamma function i.e, $\log \Gamma(x)$.

Examples

Several special values are known. For numerical integral arguments we have:

```
>>> loggamma(-2)
oo
>>> loggamma(0)
oo
>>> loggamma(1)
0
>>> loggamma(2)
0
>>> loggamma(3)
log(2)
```

and for symbolic values:

```
>>> n = Symbol('n', integer=True, positive=True)
>>> loggamma(n)
log(gamma(n))
>>> loggamma(-n)
oo
```

for half-integral values:

```
>>> loggamma(Rational(5, 2))
log(3*sqrt(pi)/4)
>>> loggamma(n/2)
log(2*(-n + 1)*sqrt(pi)*gamma(n)/gamma(n/2 + 1/2))
```

and general rational arguments:

```
>>> L = loggamma(Rational(16, 3))
>>> expand_func(L).doit()
-5*log(3) + loggamma(1/3) + log(4) + log(7) + log(10) + log(13)
>>> L = loggamma(Rational(19, 4))
>>> expand_func(L).doit()
-4*log(4) + loggamma(3/4) + log(3) + log(7) + log(11) + log(15)
>>> L = loggamma(Rational(23, 7))
>>> expand_func(L).doit()
-3*log(7) + log(2) + loggamma(2/7) + log(9) + log(16)
```

The loggamma function has the following limits towards infinity:

```
>>> loggamma(oo)
oo
>>> loggamma(-oo)
zoo
```

The loggamma function obeys the mirror symmetry if $x \in \mathbb{C} \setminus \{-\infty, 0\}$:

```
>>> c = Symbol('c', complex=True, real=False)
>>> conjugate(loggamma(c))
loggamma(conjugate(c))
```

Differentiation with respect to x is supported:

```
>>> diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```
>>> loggamma(x).series(x, 0, 4)
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + O(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> loggamma(5).evalf(30)
3.17805383034794561964694160130
>>> loggamma(I).evalf(20)
-0.65092319930185633889 - 1.8724366472624298171*I
```

See also:

gamma (page 319)

Gamma function.

lowergamma (page 326)

Lower incomplete gamma function.

uppergamma (page 325)

Upper incomplete gamma function.

polygamma (page 322)

Polygamma function.

digamma (page 324)

Digamma function.

trigamma (page 325)

Trigamma function.

diofant.functions.special.beta_functions.beta (page 328)

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Gamma_function
- <https://dlmf.nist.gov/5>
- <https://mathworld.wolfram.com/LogGammaFunction.html>
- <http://functions.wolfram.com/GammaBetaErf/LogGamma/>

class `diofant.functions.special.gamma_functions.polygamma(n, z)`

The function `polygamma(n, z)` returns `log(gamma(z)).diff($n + 1$)`.

It is a meromorphic function on \mathbb{C} and defined as the $(n+1)$ -th derivative of the logarithm of the gamma function:

$$\psi^{(n)}(z) := \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z).$$

Examples

Several special values are known:

```
>>> polygamma(0, 1)
-EulerGamma
>>> polygamma(0, Rational(1, 2))
-2*log(2) - EulerGamma
>>> polygamma(0, Rational(1, 3))
-3*log(3)/2 - sqrt(3)*pi/6 - EulerGamma
>>> polygamma(0, Rational(1, 4))
-3*log(2) - pi/2 - EulerGamma
>>> polygamma(0, 2)
-EulerGamma + 1
>>> polygamma(0, 23)
-EulerGamma + 19093197/5173168
```

```
>>> polygamma(0, oo)
oo
>>> polygamma(0, -oo)
oo
>>> polygamma(0, I*oo)
oo
>>> polygamma(0, -I*oo)
oo
```

Differentiation with respect to x is supported:

```
>>> diff(polygamma(0, x), x)
polygamma(1, x)
>>> diff(polygamma(0, x), (x, 2))
polygamma(2, x)
>>> diff(polygamma(0, x), (x, 3))
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), (x, 2))
polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
>>> diff(polygamma(2, x), (x, 2))
polygamma(4, x)
```

```
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), (x, 2))
polygamma(n + 2, x)
```

We can rewrite polygamma functions in terms of harmonic numbers:

```
>>> 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)
```

See also:

***gamma* (page 319)**

Gamma function.

***lowergamma* (page 326)**

Lower incomplete gamma function.

***uppergamma* (page 325)**

Upper incomplete gamma function.

***loggamma* (page 321)**

Log Gamma function.

***digamma* (page 324)**

Digamma function.

***trigamma* (page 325)**

Trigamma function.

***diofant.functions.special.beta_functions.beta* (page 328)**

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Polygamma_function
- <https://mathworld.wolfram.com/PolygammaFunction.html>
- <http://functions.wolfram.com/GammaBetaErf/PolyGamma/>
- <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>

`diofant.functions.special.gamma_functions.digamma(x)`

The digamma function is the first derivative of the loggamma function i.e,

$$\psi(x) := \frac{d}{dz} \log \Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}$$

In this case, `digamma(z) = polygamma(0, z)`.

See also:

***gamma* (page 319)**

Gamma function.

***lowergamma* (page 326)**

Lower incomplete gamma function.

***uppergamma* (page 325)**

Upper incomplete gamma function.

***polygamma* (page 322)**

Polygamma function.

***loggamma* (page 321)**

Log Gamma function.

***trigamma* (page 325)**

Trigamma function.

***diofant.functions.special.beta_functions.beta* (page 328)**

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Digamma_function
- <https://mathworld.wolfram.com/DigammaFunction.html>
- <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>

`diofant.functions.special.gamma_functions.trigamma(x)`

The trigamma function is the second derivative of the loggamma function i.e,

$$\psi^{(1)}(z) := \frac{d^2}{dz^2} \log \Gamma(z).$$

In this case, `trigamma(z) = polygamma(1, z)`.

See also:

`gamma` (page 319)

Gamma function.

`lowergamma` (page 326)

Lower incomplete gamma function.

`uppergamma` (page 325)

Upper incomplete gamma function.

`polygamma` (page 322)

Polygamma function.

`loggamma` (page 321)

Log Gamma function.

`digamma` (page 324)

Digamma function.

`diofant.functions.special.beta_functions.beta` (page 328)

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Trigamma_function
- <https://mathworld.wolfram.com/TrigammaFunction.html>
- <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>

class `diofant.functions.special.gamma_functions.uppergamma(a, z)`

The upper incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\Gamma(s, x) := \int_x^\infty t^{s-1} e^{-t} dt = \Gamma(s) - \gamma(s, x).$$

where $\gamma(s, x)$ is the lower incomplete gamma function, `lowergamma` (page 326). This can be shown to be the same as

$$\Gamma(s, x) = \Gamma(s) - \frac{x^s}{s} {}_1F_1 \left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x \right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt = x^{n-1} \Gamma(1-n, x).$$

Examples

```
>>> from diofant.abc import s
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
E**(-x)*x**2 + 2*E**(-x)*x + 2*E**(-x)
>>> uppergamma(-Rational(1, 2), x)
-2*sqrt(pi)*(-erf(sqrt(x)) + 1) + 2*E**(-x)/sqrt(x)
>>> uppergamma(-2, x)
expint(3, x)/x**2
```

See also:

gamma (page 319)

Gamma function.

lowergamma (page 326)

Lower incomplete gamma function.

polygamma (page 322)

Polygamma function.

loggamma (page 321)

Log Gamma function.

digamma (page 324)

Digamma function.

trigamma (page 325)

Trigamma function.

diofant.functions.special.beta_functions.beta (page 328)

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Incomplete_gamma_function#Upper_incomplete_Gamma_function
- Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- <https://dlmf.nist.gov/8>
- <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- https://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions

class diofant.functions.special.gamma_functions.lowergamma(a, x)

The lower incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\gamma(s, x) := \int_0^x t^{s-1} e^{-t} dt = \Gamma(s) - \Gamma(s, x).$$

This can be shown to be the same as

$$\gamma(s, x) = \frac{x^s}{s} {}_1F_1 \left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x \right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

Examples

```
>>> from diofant.abc import s
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
2 - E**(-x)*x**2 - 2*E**(-x)*x - 2*E**(-x)
>>> lowergamma(-Rational(1, 2), x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*E**(-x)/sqrt(x)
```

See also:

gamma (page 319)

Gamma function.

uppergamma (page 325)

Upper incomplete gamma function.

polygamma (page 322)

Polygamma function.

loggamma (page 321)

Log Gamma function.

digamma (page 324)

Digamma function.

trigamma (page 325)

Trigamma function.

diofant.functions.special.beta_functions.beta (page 328)

Euler Beta function.

References

- https://en.wikipedia.org/wiki/Incomplete_gamma_function#Lower_incomplete_Gamma_function
- Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- <https://dlmf.nist.gov/8>
- <http://functions.wolfram.com/GammaBetaErf/Gamma2/>

- <http://functions.wolfram.com/GammaBetaErf/Gamma3/>

class diofant.functions.special.beta_functions.**beta**(x, y)

The beta integral is called the Eulerian integral of the first kind by Legendre:

$$B(x, y) := \int_0^1 t^{x-1} (1-t)^{y-1} dt.$$

Beta function or Euler's first integral is closely associated with gamma function. The Beta function often used in probability theory and mathematical statistics. It satisfies properties like:

$$\begin{aligned} B(a, 1) &= \frac{1}{a} \\ B(a, b) &= B(b, a) \\ B(a, b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \end{aligned}$$

Therefore for integral values of a and b:

$$B = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

Examples

The Beta function obeys the mirror symmetry:

```
>>> conjugate(beta(x, y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both x and y is supported:

```
>>> diff(beta(x, y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)
```

```
>>> diff(beta(x, y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> beta(pi, pi).evalf(40)
0.02671848900111377452242355235388489324562
```

```
>>> beta(1 + I, 1 + I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

See also:

diofant.functions.special.gamma_functions.gamma (page 319)

Gamma function.

diofant.functions.special.gamma_functions.uppergamma (page 325)

Upper incomplete gamma function.

diofant.functions.special.gamma_functions.lowergamma (page 326)

Lower incomplete gamma function.

`diofant.functions.special.gamma_functions.polygamma` (page 322)

Polygamma function.

`diofant.functions.special.gamma_functions.loggamma` (page 321)

Log Gamma function.

`diofant.functions.special.gamma_functions.digamma` (page 324)

Digamma function.

`diofant.functions.special.gamma_functions.trigamma` (page 325)

Trigamma function.

References

- https://en.wikipedia.org/wiki/Beta_function
- <https://mathworld.wolfram.com/BetaFunction.html>
- <https://dlmf.nist.gov/5.12>

Error Functions and Fresnel Integrals

`class diofant.functions.special.error_functions.erf(arg)`

The Gauss error function. This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Examples

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:

```
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(erf(z), z)
2*E**(-z**2)/sqrt(pi)
```

We can numerically evaluate the error function to arbitrary precision on the whole complex plane:

```
>>> erf(4).evalf(30)
0.999999984582742099719981147840
```

```
>>> erf(-4*I).evalf(30)
-1296959.73071763923152794095062*I
```

See also:

erfc (page 330)

Complementary error function.

erfi (page 331)

Imaginary error function.

erf2 (page 333)

Two-argument error function.

erfinv (page 334)

Inverse error function.

erfcinv (page 335)

Inverse Complementary error function.

erf2inv (page 335)

Inverse two-argument error function.

References

- https://en.wikipedia.org/wiki/Error_function
- <https://dlmf.nist.gov/7>
- <https://mathworld.wolfram.com/Erf.html>
- <http://functions.wolfram.com/GammaBetaErf/Erf>

class diofant.functions.special.error_functions.**erfc**(arg)

Complementary Error Function. The function is defined as:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

Examples

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:

```
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(erfc(z), z)
-2*E**(-z**2)/sqrt(pi)
```

It also follows

```
>>> erfc(-z)
-erfc(z) + 2
```

We can numerically evaluate the complementary error function to arbitrary precision on the whole complex plane:

```
>>> erfc(4).evalf(30)
0.0000000154172579002800188521596734869
```

```
>>> erfc(4*I).evalf(30)
1.0 - 1296959.73071763923152794095062*I
```

See also:

erf (page 329)

Gaussian error function.

erfi (page 331)

Imaginary error function.

erf2 (page 333)

Two-argument error function.

erfinv (page 334)

Inverse error function.

erfcinv (page 335)

Inverse Complementary error function.

erf2inv (page 335)

Inverse two-argument error function.

References

- https://en.wikipedia.org/wiki/Error_function
- <https://dlmf.nist.gov/7>
- <https://mathworld.wolfram.com/Erfc.html>
- <http://functions.wolfram.com/GammaBetaErf/Erfc>

class diofant.functions.special.error_functions.**erfi**(z)

Imaginary error function. The function *erfi* is defined as:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt$$

Examples

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)
```

```
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(erfi(z), z)
2*E**(z**2)/sqrt(pi)
```

We can numerically evaluate the imaginary error function to arbitrary precision on the whole complex plane:

```
>>> erfi(2).evalf(30)
18.5648024145755525987042919132
```

```
>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

See also:

***erf* (page 329)**

Gaussian error function.

***erfc* (page 330)**

Complementary error function.

***erf2* (page 333)**

Two-argument error function.

***erfinv* (page 334)**

Inverse error function.

***erfcinv* (page 335)**

Inverse Complementary error function.

***erf2inv* (page 335)**

Inverse two-argument error function.

References

- https://en.wikipedia.org/wiki/Error_function
- <https://mathworld.wolfram.com/Erfi.html>
- <http://functions.wolfram.com/GammaBetaErf/Erfi>

class diofant.functions.special.error_functions.**erf2**(x, y)

Two-argument error function. This function is defined as:

$$\text{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} dt$$

Examples

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:

```
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to x, y is supported:

```
>>> diff(erf2(x, y), x)
-2*E**(-x**2)/sqrt(pi)
>>> diff(erf2(x, y), y)
2*E**(-y**2)/sqrt(pi)
```

See also:

erf (page 329)

Gaussian error function.

erfc (page 330)

Complementary error function.

erfi (page 331)

Imaginary error function.

erfinv (page 334)

Inverse error function.

erfcinv (page 335)

Inverse Complementary error function.

erf2inv (page 335)

Inverse two-argument error function.

References

- <http://functions.wolfram.com/GammaBetaErf/Erf2/>

class diofant.functions.special.error_functions.**erfinv**(*z*)

Inverse Error Function. The erfinv function is defined as:

$$\operatorname{erf}(x) = y \quad \Rightarrow \quad \operatorname{erfinv}(y) = x$$

Examples

Several special values are known:

```
>>> erfinv(0)
0
>>> erfinv(1)
00
```

Differentiation with respect to *x* is supported:

```
>>> diff(erfinv(x), x)
E**(erfinv(x)**2)*sqrt(pi)/2
```

We can numerically evaluate the inverse error function to arbitrary precision on [-1, 1]:

```
>>> erfinv(0.2)
0.179143454621292
```

See also:

erf (page 329)

Gaussian error function.

erfc (page 330)

Complementary error function.

erfi (page 331)

Imaginary error function.

erf2 (page 333)

Two-argument error function.

erfcinv (page 335)

Inverse Complementary error function.

erf2inv (page 335)

Inverse two-argument error function.

References

- https://en.wikipedia.org/wiki/Error_function#Inverse_functions
- <http://functions.wolfram.com/GammaBetaErf/InverseErf/>

class diofant.functions.special.error_functions.**erfcinv**(z)

Inverse Complementary Error Function. The erfcinv function is defined as:

$$\operatorname{erfc}(x) = y \quad \Rightarrow \quad \operatorname{erfcinv}(y) = x$$

Examples

Several special values are known:

```
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to x is supported:

```
>>> diff(erfcinv(x), x)
-E**(erfcinv(x)**2)*sqrt(pi)/2
```

See also:

erf (page 329)

Gaussian error function.

erfc (page 330)

Complementary error function.

erfi (page 331)

Imaginary error function.

erf2 (page 333)

Two-argument error function.

erfinv (page 334)

Inverse error function.

erf2inv (page 335)

Inverse two-argument error function.

References

- https://en.wikipedia.org/wiki/Error_function#Inverse_functions
- <http://functions.wolfram.com/GammaBetaErf/InverseErfc/>

class diofant.functions.special.error_functions.**erf2inv**(x, y)

Two-argument Inverse error function. The erf2inv function is defined as:

$$\operatorname{erf2}(x, w) = y \quad \Rightarrow \quad \operatorname{erf2inv}(x, y) = w$$

Examples

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:

```
>>> diff(erf2inv(x, y), x)
E**(-x**2 + erf2inv(x, y)**2)
>>> diff(erf2inv(x, y), y)
E**(erf2inv(x, y)**2)*sqrt(pi)/2
```

See also:

erf (page 329)

Gaussian error function.

erfc (page 330)

Complementary error function.

erfi (page 331)

Imaginary error function.

erf2 (page 333)

Two-argument error function.

erfinv (page 334)

Inverse error function.

erfcinv (page 335)

Inverse complementary error function.

References

- <http://functions.wolfram.com/GammaBetaErf/InverseErf2/>

class diofant.functions.special.error_functions.**FresnelIntegral**(z)

Base class for the Fresnel integrals.

class diofant.functions.special.error_functions.**fresnels**(z)

Fresnel integral S.

This function is defined by

$$S(z) = \int_0^z \sin \frac{\pi}{2} t^2 dt.$$

It is an entire function.

Examples

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 $\overline{S(z)} = S(\bar{z})$:

```
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> 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
```

See also:

[fresnelc](#) (page 338)

Fresnel cosine integral.

References

- https://en.wikipedia.org/wiki/Fresnel_integral
- <https://dlmf.nist.gov/7>
- <https://mathworld.wolfram.com/FresnelIntegrals.html>
- <http://functions.wolfram.com/GammaBetaErf/FresnelS>
- The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley

class diofant.functions.special.error_functions.fresnelc(*z*)

Fresnel integral C.

This function is defined by

$$C(z) = \int_0^z \cos \frac{\pi}{2} t^2 dt.$$

It is an entire function.

Examples

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 $\overline{C(z)} = C(\bar{z})$:

```
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to *z* is supported:

```
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> 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
```

See also:

fresnels (page 336)

Fresnel sine integral.

References

- https://en.wikipedia.org/wiki/Fresnel_integral
- <https://dlmf.nist.gov/7>
- <https://mathworld.wolfram.com/FresnelIntegrals.html>
- <http://functions.wolfram.com/GammaBetaErf/FresnelC>
- The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley

Exponential, Logarithmic and Trigonometric Integrals

class diofant.functions.special.error_functions.**Ei**(z)

The classical exponential integral.

For use in Diofant, this function is defined as

$$\text{Ei}(x) = \sum_{n=1}^{\infty} \frac{x^n}{n n!} + \log(x) + \gamma,$$

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:

$$\text{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt$$

If the integral is interpreted as a Cauchy principal value, this statement holds for $x > 0$ and $\text{Ei}(x)$ as defined above.

Note that we carefully avoided defining $\text{Ei}(x)$ for negative real x . This is because above integral formula does not hold for any polar lift of such x , indeed all branches of $\text{Ei}(x)$ above the negative reals are imaginary.

However, the following statement holds for all $x \in \mathbb{R}^*$:

$$\int_{-\infty}^x \frac{e^t}{t} dt = \frac{\text{Ei}(|x|e^{i \arg(x)}) + \text{Ei}(|x|e^{-i \arg(x)})}{2},$$

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 .

Examples

The exponential integral in Diofant 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).evalf(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```
>>> Ei(polar_lift(-1)).evalf(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)
E**x/x
```

The exponential integral is related to many other special functions. For example:

```
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)
```

See also:

[expint](#) (page 341)

Generalized exponential integral.

[E1](#) (page 342)

Special case of the generalized exponential integral.

[li](#) (page 343)

Logarithmic integral.

[Li](#) (page 344)

Offset logarithmic integral.

[Si](#) (page 346)

Sine integral.

[Ci](#) (page 347)

Cosine integral.

[Shi](#) (page 348)

Hyperbolic sine integral.

[Chi](#) (page 349)

Hyperbolic cosine integral.

[diofant.functions.special.gamma_functions.uppergamma](#) (page 325)

Upper incomplete gamma function.

References

- <https://dlmf.nist.gov/6.6>
- https://en.wikipedia.org/wiki/Exponential_integral
- Abramowitz & Stegun, section 5: http://people.math.sfu.ca/~cbm/aands/page_228.htm

class diofant.functions.special.error_functions.expint(nu, z)

Generalized exponential integral.

This function is defined as

$$E_{\nu}(z) = z^{\nu-1} \Gamma(1 - \nu, z),$$

where $\Gamma(1 - \nu, z)$ is the upper incomplete gamma function (uppergamma).

Hence for z with positive real part we have

$$E_{\nu}(z) = \int_1^{\infty} \frac{e^{-zt}}{t^{\nu}} dt,$$

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for $E_{\nu}(z)$. If ν is a non-positive integer the exponential integral is thus an unbranched function of z , otherwise there is a branch point at the origin. Refer to the incomplete gamma function documentation for details of the branching behavior.

Examples

```
>>> from diofant.abc import nu
```

Differentiation is supported. Differentiation with respect to z explains further the name: for integral orders, the exponential integral is an iterated integral of the exponential function.

```
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to ν has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg(((), (1, 1)), ((0, 0, -nu + 1), ()), z)
```

At non-positive integer orders, the exponential integral reduces to the exponential function:

```
>>> expint(0, z)
E**(-z)/z
>>> expint(-1, z)
E**(-z)/z + E**(-z)/z**2
```

At half-integers it reduces to error functions:

```
>>> expint(Rational(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:

```
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + E**(-z)*(-z**3 + z**2 - 2*z + 6)/24
```

The generalized exponential integral is essentially equivalent to the incomplete gamma function:

```
>>> expint(nu, z).rewrite(uppergamma)
z**(nu - 1)*uppergamma(-nu + 1, z)
```

As such it is branched at the origin:

```
>>> 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)*(E**(2*I*pi*nu) - 1)*gamma(-nu + 1) + expint(nu, z)
```

See also:

Ei (page 339)

Another related function called exponential integral.

E1 (page 342)

The classical case, returns `expint(1, z)`.

li (page 343)

Logarithmic integral.

Li (page 344)

Offset logarithmic integral.

Si (page 346)

Sine integral.

Ci (page 347)

Cosine integral.

Shi (page 348)

Hyperbolic sine integral.

Chi (page 349)

Hyperbolic cosine integral.

diofant.functions.special.gamma_functions.uppergamma (page 325)

References

- <https://dlmf.nist.gov/8.19>
- <http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/>
- https://en.wikipedia.org/wiki/Exponential_integral

`diofant.functions.special.error_functions.E1(z)`

Classical case of the generalized exponential integral.

This is equivalent to `expint(1, z)`.

See also:

Ei (page 339)

Exponential integral.

expint (page 341)

Generalized exponential integral.

li (page 343)

Logarithmic integral.

Li (page 344)

Offset logarithmic integral.

Si (page 346)

Sine integral.

Ci (page 347)

Cosine integral.

Shi (page 348)

Hyperbolic sine integral.

Chi (page 349)

Hyperbolic cosine integral.

class diofant.functions.special.error_functions.**li**(z)

The classical logarithmic integral.

For the use in Diofant, this function is defined as

$$\operatorname{li}(x) = \int_0^x \frac{1}{\log(t)} dt.$$

Examples

Several special values are known:

```
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to z is supported:

```
>>> 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:

```
>>> li(z).rewrite(Ei)
Ei(log(z))
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> li(2).evalf(30)
1.04516378011749278484458888919
```

```
>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```
>>> from mpmath import findroot
>>> print(findroot(li, 2))
1.45136923488338
```

Further transformations include rewriting *li* in terms of the trigonometric integrals *Si*, *Ci*, *Shi* and *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))
```

See also:

***Li* (page 344)**

Offset logarithmic integral.

***Ei* (page 339)**

Exponential integral.

***expint* (page 341)**

Generalized exponential integral.

***E1* (page 342)**

Special case of the generalized exponential integral.

***Si* (page 346)**

Sine integral.

***Ci* (page 347)**

Cosine integral.

***Shi* (page 348)**

Hyperbolic sine integral.

***Chi* (page 349)**

Hyperbolic cosine integral.

References

- https://en.wikipedia.org/wiki/Logarithmic_integral
- <https://mathworld.wolfram.com/LogarithmicIntegral.html>
- <https://dlmf.nist.gov/6>
- <https://mathworld.wolfram.com/SoldnersConstant.html>

class diofant.functions.special.error_functions.**Li**(*z*)

The offset logarithmic integral.

For the use in Diofant, this function is defined as

$$\text{Li}(x) = \text{li}(x) - \text{li}(2)$$

Examples

The following special value is known:

```
>>> Li(2)
0
```

Differentiation with respect to z is supported:

```
>>> diff(Li(z), z)
1/log(z)
```

The shifted logarithmic integral can be written in terms of $li(z)$:

```
>>> 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
```

See also:

li (page 343)

Logarithmic integral.

Ei (page 339)

Exponential integral.

expint (page 341)

Generalized exponential integral.

E1 (page 342)

Special case of the generalized exponential integral.

Si (page 346)

Sine integral.

Ci (page 347)

Cosine integral.

Shi (page 348)

Hyperbolic sine integral.

Chi (page 349)

Hyperbolic cosine integral.

References

- https://en.wikipedia.org/wiki/Logarithmic_integral
- <https://mathworld.wolfram.com/LogarithmicIntegral.html>
- <https://dlmf.nist.gov/6>

class diofant.functions.special.error_functions.**Si**(z)

Sine integral.

This function is defined by

$$\text{Si}(z) = \int_0^z \frac{\sin t}{t} dt.$$

It is an entire function.

Examples

The sine integral is an antiderivative of $\sin(z)/z$:

```
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```
>>> 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:

```
>>> 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
```

See also:

Ci (page 347)

Cosine integral.

Shi (page 348)

Hyperbolic sine integral.

Chi (page 349)

Hyperbolic cosine integral.

Ei (page 339)

Exponential integral.

expint (page 341)

Generalized exponential integral.

E1 (page 342)

Special case of the generalized exponential integral.

li (page 343)

Logarithmic integral.

Li (page 344)

Offset logarithmic integral.

References

- https://en.wikipedia.org/wiki/Trigonometric_integral

class diofant.functions.special.error_functions.**Ci**(z)

Cosine integral.

This function is defined for positive x by

$$\text{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt = - \int_x^\infty \frac{\cos t}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Ci}(z) = - \frac{E_1(e^{i\pi/2}z) + E_1(e^{-i\pi/2}z)}{2}$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for $z \in \mathbb{C}$ with $\Re(z) > 0$. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

Examples

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:

```
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary \cos under multiplication by i :

```
>>> 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:

```
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```

See also:

Si (page 346)

Sine integral.

Shi (page 348)

Hyperbolic sine integral.

Chi (page 349)

Hyperbolic cosine integral.

Ei (page 339)

Exponential integral.

expint (page 341)

Generalized exponential integral.

E1 (page 342)

Special case of the generalized exponential integral.

li (page 343)

Logarithmic integral.

Li (page 344)

Offset logarithmic integral.

References

- https://en.wikipedia.org/wiki/Trigonometric_integral

class diofant.functions.special.error_functions.**Shi**(*z*)

Sinh integral.

This function is defined by

$$\text{Shi}(z) = \int_0^z \frac{\sinh t}{t} dt.$$

It is an entire function.

Examples

The Sinh integral is a primitive of $\sinh(z)/z$:

```
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```
>>> 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:


```
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

See also:

Si (page 346)

Sine integral.

Ci (page 347)

Cosine integral.

Chi (page 349)

Hyperbolic cosine integral.

Ei (page 339)

Exponential integral.

expint (page 341)

Generalized exponential integral.

E1 (page 342)

Special case of the generalized exponential integral.

li (page 343)

Logarithmic integral.

Li (page 344)

Offset logarithmic integral.

References

- https://en.wikipedia.org/wiki/Trigonometric_integral

class diofant.functions.special.error_functions.**Chi**(z)

Cosh integral.

This function is defined for positive x by

$$\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Chi}(z) = \text{Ci}\left(e^{i\pi/2}z\right) - i\frac{\pi}{2},$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

Examples

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:

```
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```

The cosh integral behaves somewhat like ordinary cosh under multiplication by i :

```
>>> 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:

```
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

See also:

Si (page 346)

Sine integral.

Ci (page 347)

Cosine integral.

Shi (page 348)

Hyperbolic sine integral.

Ei (page 339)

Exponential integral.

expint (page 341)

Generalized exponential integral.

E1 (page 342)

Special case of the generalized exponential integral.

li (page 343)

Logarithmic integral.

Li (page 344)

Offset logarithmic integral.

References

- https://en.wikipedia.org/wiki/Trigonometric_integral

Bessel Type Functions

class diofant.functions.special.bessel.BesselBase(*nu*, *z*)

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 $2 * F_n' = -_a * F_{n+1} + _b * F_{n-1}$.

property argument

The argument of the bessel-type function.

property order

The order of the bessel-type function.

class diofant.functions.special.bessel.besselj(*nu*, *z*)

Bessel function of the first kind.

The Bessel J function of order ν is defined to be the function satisfying Bessel’s differential equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2)w = 0,$$

with Laurent expansion

$$J_\nu(z) = z^\nu \left(\frac{1}{\Gamma(\nu + 1)2^\nu} + O(z^2) \right),$$

if ν is not a negative integer. If $\nu = -n \in \mathbb{Z}_{<0}$ is a negative integer, then the definition is

$$J_{-n}(z) = (-1)^n J_n(z).$$

Examples

Create a Bessel function object:

```
>>> 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
```

See also:

[*bessely*](#) (page 352), [*besseli*](#) (page 352), [*besselk*](#) (page 353)

References

- Abramowitz, Milton; Stegun, Irene A., eds. (1965), “Chapter 9”, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- https://en.wikipedia.org/wiki/Bessel_function
- <http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/>

class diofant.functions.special.bessel.**bessely**(*nu*, *z*)

Bessel function of the second kind.

The Bessel Y function of order ν is defined as

$$Y_{\nu}(z) = \lim_{\mu \rightarrow \nu} \frac{J_{\mu}(z) \cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

where $J_{\mu}(z)$ is the Bessel function of the first kind.

It is a solution to Bessel’s equation, and linearly independent from J_{ν} .

Examples

```
>>> 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)
```

See also:

[*besselj*](#) (page 351), [*besseli*](#) (page 352), [*besselk*](#) (page 353)

References

- <http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/>

class diofant.functions.special.bessel.**besseli**(*nu*, *z*)

Modified Bessel function of the first kind.

The Bessel I function is a solution to the modified Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 + \nu^2)w = 0.$$

It can be defined as

$$I_{\nu}(z) = i^{-\nu} J_{\nu}(iz),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind.

Examples

```
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(n + 1, z)/2
```

See also:

[besselj](#) (page 351), [bessely](#) (page 352), [besselk](#) (page 353)

References

- <http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/>

class diofant.functions.special.bessel.besselk(*nu*, *z*)

Modified Bessel function of the second kind.

The Bessel K function of order ν is defined as

$$K_{\nu}(z) = \lim_{\mu \rightarrow \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_{\mu}(z)}{\sin(\pi\mu)},$$

where $I_{\mu}(z)$ is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from Y_{ν} .

Examples

```
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

See also:

[besselj](#) (page 351), [besseli](#) (page 352), [bessely](#) (page 352)

References

- <http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/>

class diofant.functions.special.bessel.hankel1(*nu*, *z*)

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.

Examples

```
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

See also:

[hankel2](#) (page 354), [besselj](#) (page 351), [bessely](#) (page 352)

References

- <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/>

class diofant.functions.special.bessel.hankel2(*nu*, *z*)

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 $H_{\nu}^{(1)}$.

Examples

```
>>> hankel2(n, z).diff(z)
hankel2(n - 1, z)/2 - hankel2(n + 1, z)/2
```

See also:

[hankel1](#) (page 353), [besselj](#) (page 351), [bessely](#) (page 352)

References

- <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/>

class diofant.functions.special.bessel.jn(*nu*, *z*)

Spherical Bessel function of the first kind.

This function is a solution to the spherical Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + 2z \frac{dw}{dz} + (z^2 - \nu(\nu + 1))w = 0.$$

It can be defined as

$$j_{\nu}(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+\frac{1}{2}}(z),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind.

Examples

```
>>> 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 [`diofant.polys.orthopolys.spherical_bessel_fn\(\)`](#) (page 543).

See also:

[`besselj`](#) (page 351), [`bessely`](#) (page 352), [`besselk`](#) (page 353), [`yn`](#) (page 355)

class `diofant.functions.special.bessel.yn(nu, z)`

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.

Examples

```
>>> 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:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

See also:

[`besselj`](#) (page 351), [`bessely`](#) (page 352), [`besselk`](#) (page 353), [`jn`](#) (page 354)

class `diofant.functions.special.bessel.jn_zeros(n, k, dps=15)`

Zeros of the spherical Bessel function of the first kind.

This returns an array of zeros of j_n up to the k -th zero.

Examples

```
>>> jn.zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

See also:

[jn](#) (page 354), [yn](#) (page 355), [besselj](#) (page 351), [besselk](#) (page 353), [bessely](#) (page 352)

Airy Functions

class diofant.functions.special.bessel.**AiryBase**(*args)

Abstract base class for Airy functions.

This class is meant to reduce code duplication.

class diofant.functions.special.bessel.**airyai**(arg)

The Airy function Ai of the first kind.

The Airy function $\text{Ai}(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{d^2 w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$\text{Ai}(z) := \frac{1}{\pi} \int_0^\infty \cos\left(\frac{t^3}{3} + zt\right) dt.$$

Examples

Create an Airy function object:

```
>>> airyai(z)
airyai(z)
```

Several special values are known:

```
>>> airyai(0)
root(3, 3)/(3*gamma(2/3))
>>> airyai(oo)
0
>>> airyai(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), (z, 2))
z*airyai(z)
```

Series expansion is also supported:


```
>>> airyai(z).series(z, 0, 3)
root(3, 6)**5*gamma(1/3)/(6*pi) - root(3, 6)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyai(-2).evalf(50)
0.22740742820168557599192443603787379946077222541710
```

Rewrite $\text{Ai}(z)$ in terms of hypergeometric functions:

```
>>> airyai(z).rewrite(hyper)
-root(3, 3)**2*z*hyper((), (4/3,), z**3/9)/(3*gamma(1/3)) + root(3, 3)*hyper((), (2/3,), z**3/9)/(3*gamma(2/3))
```

See also:

***airybi* (page 357)**

Airy function of the second kind.

***airyaiprime* (page 359)**

Derivative of the Airy function of the first kind.

***airybiprime* (page 360)**

Derivative of the Airy function of the second kind.

References

- https://en.wikipedia.org/wiki/Airy_function
- <https://dlmf.nist.gov/9>
- https://www.encyclopediaofmath.org/index.php/Airy_functions
- <https://mathworld.wolfram.com/AiryFunctions.html>

class diofant.functions.special.bessel.**airybi**(arg)

The Airy function Bi of the second kind.

The Airy function $\text{Bi}(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{d^2 w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$\text{Bi}(z) := \frac{1}{\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} + zt\right) + \sin\left(\frac{t^3}{3} + zt\right) dt.$$

Examples

Create an Airy function object:

```
>>> airybi(z)
airybi(z)
```

Several special values are known:

```
>>> airybi(0)
root(3, 6)**5/(3*gamma(2/3))
>>> airybi(oo)
oo
>>> airybi(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), (z, 2))
z*airybi(z)
```

Series expansion is also supported:

```
>>> airybi(z).series(z, 0, 3)
root(3, 3)*gamma(1/3)/(2*pi) + root(3, 3)**2*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite $\text{Bi}(z)$ in terms of hypergeometric functions:

```
>>> airybi(z).rewrite(hyper)
root(3, 6)*z*hyper([], (4/3,), z**3/9)/gamma(1/3) + root(3, 6)**5*hyper([], (2/3,
↪), z**3/9)/(3*gamma(2/3))
```

See also:

[airyai](#) (page 356)

Airy function of the first kind.

[airyaiprime](#) (page 359)

Derivative of the Airy function of the first kind.

[airybiprime](#) (page 360)

Derivative of the Airy function of the second kind.

References

- https://en.wikipedia.org/wiki/Airy_function
- <https://dlmf.nist.gov/9>
- https://www.encyclopediaofmath.org/index.php/Airy_functions
- <https://mathworld.wolfram.com/AiryFunctions.html>

class diofant.functions.special.bessel.airyaiprime(*arg*)

The derivative Ai' of the Airy function of the first kind.

The Airy function $Ai'(z)$ is defined to be the function

$$Ai'(z) := \frac{d Ai(z)}{dz}.$$

Examples

Create an Airy function object:

```
>>> airyaiprime(z)
airyaiprime(z)
```

Several special values are known:

```
>>> airyaiprime(0)
-root(3, 3)**2/(3*gamma(1/3))
>>> airyaiprime(oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> conjugate(airyaiprime(z))
airyaiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(airyaiprime(z), z)
z*airyai(z)
>>> diff(airyaiprime(z), (z, 2))
z*airyaiprime(z) + airyai(z)
```

Series expansion is also supported:

```
>>> airyaiprime(z).series(z, 0, 3)
-root(3, 3)**2/(3*gamma(1/3)) + root(3, 3)*z**2/(6*gamma(2/3)) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyaiprime(-2).evalf(50)
0.61825902074169104140626429133247528291577794512415
```

Rewrite $Ai'(z)$ in terms of hypergeometric functions:

```
>>> airyaiprime(z).rewrite(hyper)
root(3, 3)*z**2*hyper(( ), (5/3, ), z**3/9)/(6*gamma(2/3)) - root(3, 3)**2*hyper(( ),
↪ (1/3, ), z**3/9)/(3*gamma(1/3))
```

See also:

***airyai* (page 356)**

Airy function of the first kind.

***airybi* (page 357)**

Airy function of the second kind.

***airybiprime* (page 360)**

Derivative of the Airy function of the second kind.

References

- https://en.wikipedia.org/wiki/Airy_function
- <https://dlmf.nist.gov/9>
- https://www.encyclopediaofmath.org/index.php/Airy_functions
- <https://mathworld.wolfram.com/AiryFunctions.html>

class diofant.functions.special.bessel.**airybiprime**(arg)

The derivative Bi' of the Airy function of the first kind.

The Airy function $\text{Bi}'(z)$ is defined to be the function

$$\text{Bi}'(z) := \frac{d \text{Bi}(z)}{dz}.$$

Examples

Create an Airy function object:

```
>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```
>>> airybiprime(0)
root(3, 6)/gamma(1/3)
>>> airybiprime(oo)
oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> diff(airybiprime(z), z)
z*airybi(z)
>>> diff(airybiprime(z), (z, 2))
z*airybiprime(z) + airybi(z)
```

Series expansion is also supported:

```
>>> airybiprime(z).series(z, 0, 3)
root(3, 6)/gamma(1/3) + root(3, 6)**5*z**2/(6*gamma(2/3)) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybiprime(-2).evalf(50)
0.27879516692116952268509756941098324140300059345163
```

Rewrite $\text{Bi}'(z)$ in terms of hypergeometric functions:

```
>>> airybiprime(z).rewrite(hyper)
root(3, 6)**5*z**2*hyper(( ), (5/3, ), z**3/9)/(6*gamma(2/3)) + root(3, 6)*hyper(( ),
↳ (1/3, ), z**3/9)/gamma(1/3)
```

See also:

***airyai* (page 356)**

Airy function of the first kind.

***airybi* (page 357)**

Airy function of the second kind.

***airyaiprime* (page 359)**

Derivative of the Airy function of the first kind.

References

- https://en.wikipedia.org/wiki/Airy_function
- <https://dlmf.nist.gov/9>
- https://www.encyclopediaofmath.org/index.php/Airy_functions
- <https://mathworld.wolfram.com/AiryFunctions.html>

B-Splines

`diofant.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 . 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:

```
>>> d = 0
>>> knots = range(5)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, (x >= 0) & (x <= 1)), (0, true))
```

For a given (d , $knots$) there are $\text{len}(knots) - d - 1$ B-splines defined, that are indexed by n (starting at 0).

Here is an example of a cubic B-spline:

```
>>> bspline_basis(3, range(5), 0, x)
Piecewise((x**3/6, (x >= 0) & (x < 1)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3,
           (x >= 1) & (x < 2)),
          (x**3/2 - 4*x**2 + 10*x - 22/3,
           (x >= 2) & (x < 3)),
          (-x**3/6 + 2*x**2 - 8*x + 32/3,
           (x >= 3) & (x <= 4)),
          (0, true))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```
>>> d = 1
>>> knots = [0, 0, 2, 3, 4]
>>> bspline_basis(d, knots, 0, x)
Piecewise((-x/2 + 1, (x >= 0) & (x <= 2)), (0, true))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-splines many times, it is best to lambdify them first:

```
>>> d = 3
>>> knots = range(10)
>>> b0 = bspline_basis(d, knots, 0, x)
>>> f = lambdify(x, b0)
>>> y = f(0.5)
```

See also:

diofant.functions.special.bsplines.bspline_basis_set (page 362)

References

- <https://en.wikipedia.org/wiki/B-spline>

`diofant.functions.special.bsplines.bspline_basis_set(d, knots, x)`

Return the $\text{len}(\text{knots}) - d - 1$ B-splines at x of degree d with knots.

This function returns a list of Piecewise polynomials that are the $\text{len}(\text{knots}) - d - 1$ B-splines of degree d for the given knots. This function calls `bspline_basis(d, knots, n, x)` for different values of n .

Examples

```
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines
[Piecewise((x**2/2, (x >= 0) & (x < 1)),
            (-x**2 + 3*x - 3/2, (x >= 1) & (x < 2)),
            (x**2/2 - 3*x + 9/2, (x >= 2) & (x <= 3)),
            (0, true)),
 Piecewise((x**2/2 - x + 1/2, (x >= 1) & (x < 2)),
            (-x**2 + 5*x - 11/2, (x >= 2) & (x < 3)),
            (x**2/2 - 4*x + 8, (x >= 3) & (x <= 4)),
            (0, true))]
```

See also:

diofant.functions.special.bsplines.bspline_basis (page 361)

Riemann Zeta and Related Functions

class diofant.functions.special.zeta_functions.zeta(*z*, *a=None*)

Hurwitz zeta function (or Riemann zeta function).

For $\operatorname{Re}(a) > 0$ and $\operatorname{Re}(s) > 1$, 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 $\operatorname{Re}(a) > 0$ 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 $\operatorname{Re}(a) < 0$), see the documentation of [lerchphi](#) (page 366) 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 364), [lerchphi](#) (page 366), [polylog](#) (page 365)

References

- <https://dlmf.nist.gov/25.11>
- https://en.wikipedia.org/wiki/Hurwitz_zeta_function

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 diofant.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:

```
>>> 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).evalf()
1.20205690315959
```

The derivative of $\zeta(s, a)$ with respect to a is easily computed:

```
>>> 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, [*diofant.functions.special.zeta_functions.lerchphi*](#) (page 366):

```
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

class `diofant.functions.special.zeta_functions.dirichlet_eta(s)`

Dirichlet eta function.

For $\operatorname{Re}(s) > 0$, this function is defined as

$$\eta(s) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n^s}.$$

It admits a unique analytic continuation to all of \mathbb{C} . It is an entire, unbranched function.

See also:

[*zeta*](#) (page 363)

References

- https://en.wikipedia.org/wiki/Dirichlet_eta_function
- <https://mathworld.wolfram.com/DirichletEtaFunction.html>

Examples

The Dirichlet eta function is closely related to the Riemann zeta function:

```
>>> from diofant.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(-2*(-s + 1) + 1)*zeta(s)
```

class diofant.functions.special.zeta_functions.polylog(*s*, *z*)

Polylogarithm function.

For $|z| < 1$ and $s \in \mathbb{C}$, the polylogarithm is defined by

$$\text{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for n . It admits an analytic continuation which is branched at $z = 1$ (notably not on the sheet of initial definition), $z = 0$ and $z = \infty$.

The name polylogarithm comes from the fact that for $s = 1$, the polylogarithm is related to the ordinary logarithm (see examples), and that

$$\text{Li}_{s+1}(z) = \int_0^z \frac{\text{Li}_s(t)}{t} dt.$$

The polylogarithm is a special case of the Lerch transcendent:

$$\text{Li}_s(z) = z\Phi(z, s, 1)$$

See also:

[zeta](#) (page 363), [lerchphi](#) (page 366)

Examples

For $z \in \{0, 1, -1\}$, the polylogarithm is automatically expressed using other functions:

```
>>> from diofant.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()`:

```
>>> expand_func(polylog(1, z))
-log(-z + 1)
>>> expand_func(polylog(0, z))
z/(-z + 1)
```

The derivative with respect to z can be computed in closed form:

```
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

References

- <https://en.wikipedia.org/wiki/Polylogarithm>
- <https://mathworld.wolfram.com/Polylogarithm.html>

class diofant.functions.special.zeta_functions.lerchphi(*args)

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 $\operatorname{Re}(a) \leq 0$.

Assume now $\operatorname{Re}(a) > 0$. The integral representation

$$\Phi_0(z, s, a) = \int_0^{\infty} \frac{t^{s-1} e^{-at}}{1 - ze^{-t}} \frac{dt}{\Gamma(s)}$$

provides an analytic continuation to $\mathbb{C} - [1, \infty)$. Finally, for $x \in (1, \infty)$ we find

$$\lim_{\epsilon \rightarrow 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \rightarrow 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},$$

using the standard branch for both $\log x$ and $\log \log x$ (a branch of $\log \log x$ is needed to evaluate $\log x^{s-1}$). This concludes the analytic continuation. The Lerch transcendent is thus branched at $z \in \{0, 1, \infty\}$ and $a \in \mathbb{Z}_{\leq 0}$. For fixed z, a outside these branch points, it is an entire function of s .

See also:

polylog (page 365), *zeta* (page 363)

References

- Bateman, H.; Erdélyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill. Section 1.11.
- <https://dlmf.nist.gov/25.14>
- https://en.wikipedia.org/wiki/Lerch_transcendent

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 diofant.abc import s
>>> 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:

```
>>> expand_func(lerchphi(z, s, Rational(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, Rational(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 `diofant.functions.special.hyper.hyper(a_p, b_q, z)`

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 $p \leq q$, and thus defines an entire single-valued function in this case. If $p = q + 1$ the series converges for $|z| < 1$, and can be continued analytically into a half-plane. If $p > q + 1$ the series is divergent for all z .

Note: The hypergeometric function constructor currently does *not* check if the parameters actually yield a well-defined function.

Examples

The parameters a_p and b_q can be passed as arbitrary iterables, for example:

```
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing:

```
>>> pprint(hyper((1, 2, 3), [3, 4], x))

$${}_3F_2 \left( \begin{matrix} 1, 2, 3 \\ 3, 4 \end{matrix} \middle| x \right)$$

```

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:

```
>>> hyperexpand(hyper([], [], x))
E**x
```

You can also use `expand_func`:

```
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```
>>> hyperexpand(hyper([], [Rational(1, 2)], -x**2/4))
cos(x)
>>> hyperexpand(x*hyper([Rational(1, 2), Rational(1, 2)], [Rational(3, 2)], x**2))
asin(x)
```

We can also sometimes hyperexpand parametric functions:

```
>>> hyperexpand(hyper([-a], [], x))
(-x + 1)**a
```

See also:

`diofant.simplify.hyperexpand` (page 603), `diofant.functions.special.gamma_functions.gamma` (page 319), `diofant.functions.special.hyper.meijerg` (page 369)

References

- Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- https://en.wikipedia.org/wiki/Generalized_hypergeometric_function

property `ap`

Numerator parameters of the hypergeometric function.

property `argument`

Argument of the hypergeometric function.

property `bq`

Denominator parameters of the hypergeometric function.

property `convergence_statement`

Return a condition on z under which the series converges.

property `eta`

A quantity related to the convergence of the series.

property `radius_of_convergence`

Compute the radius of convergence of the defining series.

Note that even if this is not ∞ , the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

```
>>> 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 `diofant.functions.special.hyper.meijerg(*args)`

The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.

The Meijer G-function depends on four sets of parameters. There are “*numerator parameters*” a_1, \dots, a_n and a_{n+1}, \dots, a_p , and there are “*denominator parameters*” b_1, \dots, b_m

and b_{m+1}, \dots, b_q . Confusingly, it is traditionally denoted as follows (note the position of m, n, p, q , and how they relate to the lengths of the four parameter vectors):

$$G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_n & a_{n+1}, \dots, a_p \\ b_1, \dots, b_m & b_{m+1}, \dots, b_q \end{matrix} \middle| z \right).$$

However, in diofant the four parameter vectors are always available separately (see examples), so that there is no need to keep track of the decorating sub- and super-scripts on the G symbol.

The G function is defined as the following integral:

$$\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where $\Gamma(z)$ is the gamma function. There are three possible contours which we will not describe in detail here (see the references). If the integral converges along more than one of them the definitions agree. The contours all separate the poles of $\Gamma(1 - a_j + s)$ from the poles of $\Gamma(b_k - s)$, so in particular the G function is undefined if $a_j - b_k \in \mathbb{Z}_{>0}$ for some $j \leq n$ and $k \leq m$.

The conditions under which one of the contours yields a convergent integral are complicated and we do not state them here, see the references.

Note: Currently the Meijer G-function constructor does *not* check any convergence conditions.

Examples

You can pass the parameters either as four separate vectors:

```
>>> pprint(meijerg([1, 2], [a, 4], [5], [], x))
G
1, 2 (1, 2 a, 4 | x)
4, 1 (5
```

or as two nested vectors:

```
>>> pprint(meijerg(([1, 2], [3, 4]), ([5], []), x))
G
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:

```
>>> g = meijerg([1], [2], [3], [4], x)
>>> pprint(g)
G
1, 1 (1 2 | x)
2, 2 (3 4 | x)
>>> g.an
(1,)
>>> g.ap
(1, 2)
```

(continues on next page)

(continued from previous page)

```
>>> 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:

```
>>> 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:

```
>>> expand_func(meijerg([], [], [[0], []], -x))
E**x
>>> hyperexpand(meijerg([], [], [[Rational(1, 2)], [0]], (x/2)**2))
sin(x)/sqrt(pi)
```

See also:

[*diofant.functions.special.hyper.hyper*](#) (page 367), [*diofant.simplify.hyperexpand*](#) (page 603)

References

- Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- https://en.wikipedia.org/wiki/Meijer_G-function

property an

First set of numerator parameters.

property aother

Second set of numerator parameters.

property ap

Combined numerator parameters.

property argument

Argument of the Meijer G-function.

property bm

First set of denominator parameters.

property bother

Second set of denominator parameters.

property bq

Combined denominator parameters.

property delta

A quantity related to the convergence region of the integral, c.f. references.

get_period()

Return a number P such that $G(x*\exp(I*P)) == G(x)$.

```
>>> 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, Rational(1, 2), Rational(1, 3)], [1], z).get_
period()
12*pi
```

integrand(s)

Get the defining integrand $D(s)$.

property nu

A quantity related to the convergence region of the integral, c.f. references.

Elliptic integrals**class** diofant.functions.special.elliptic_integrals.elliptic_k(m)

The complete elliptic integral of the first kind, defined by

$$K(m) = F\left(\frac{\pi}{2} \middle| m\right)$$

where $F(z|m)$ is the Legendre incomplete elliptic integral of the first kind.

The function $K(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Examples

```
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(m).series(m, n=3)
pi/2 + pi*m/8 + 9*pi*m**2/128 + 0(m**3)
```

References

- https://en.wikipedia.org/wiki/Elliptic_integrals
- <http://functions.wolfram.com/EllipticIntegrals/EllipticK>

See also:

[*elliptic_f*](#) (page 372)

class diofant.functions.special.elliptic_integrals.elliptic_f(z, m)

The Legendre incomplete elliptic integral of the first kind, defined by

$$F(z|m) = \int_0^z \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

This function reduces to a complete elliptic integral of the first kind, $K(m)$, when $z = \pi/2$.

Examples

```
>>> 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
```

References

- https://en.wikipedia.org/wiki/Elliptic_integrals
- <http://functions.wolfram.com/EllipticIntegrals/EllipticF>

See also:

`elliptic_k` (page 372)

class diofant.functions.special.elliptic_integrals.elliptic_e(*z*, *m=None*)

Called with two arguments *z* and *m*, evaluates the incomplete elliptic integral of the second kind, defined by

$$E(z|m) = \int_0^z \sqrt{1 - m \sin^2 t} dt$$

Called with a single argument *m*, evaluates the Legendre complete elliptic integral of the second kind

$$E(m) = E\left(\frac{\pi}{2}|m\right)$$

The function $E(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Examples

```
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + O(z**6)
>>> elliptic_e(m).series(m, n=4)
pi/2 - pi*m/8 - 3*pi*m**2/128 - 5*pi*m**3/512 + O(m**4)
>>> elliptic_e(1 + I, 2 - I/2).evalf()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I
```

References

- https://en.wikipedia.org/wiki/Elliptic_integrals
- <http://functions.wolfram.com/EllipticIntegrals/EllipticE2>
- <http://functions.wolfram.com/EllipticIntegrals/EllipticE>

class diofant.functions.special.elliptic_integrals.elliptic_pi(*n*, *m*, *z=None*)

Called with three arguments *n*, *z* and *m*, evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$\Pi(n; z|m) = \int_0^z \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}}$$

Called with two arguments *n* and *m*, evaluates the complete elliptic integral of the third kind:

$$\Pi(n|m) = \Pi\left(n; \frac{\pi}{2} |m\right)$$

Examples

```
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + O(z**4)
>>> elliptic_pi(0.5 + I, 1.0 - I, 1.2)
2.50232379629182 - 0.760939574180767*I
>>> elliptic_pi(0, 0)
pi/2
>>> elliptic_pi(1.0 - I/3, 2.0 + I)
3.29136443417283 + 0.32555634906645*I
```

References

- https://en.wikipedia.org/wiki/Elliptic_integrals
- <http://functions.wolfram.com/EllipticIntegrals/EllipticPi3>
- <http://functions.wolfram.com/EllipticIntegrals/EllipticPi>

Orthogonal Polynomials

This module mainly implements special orthogonal polynomials.

See also functions.combinatorial.numbers which contains some combinatorial polynomials.

Jacobi Polynomials

class diofant.functions.special.polynomials.jacobi(*n*, *a*, *b*, *x*)

Jacobi polynomial $P_n^{(\alpha, \beta)}(x)$

jacobi(*n*, *alpha*, *beta*, *x*) gives the *n*th Jacobi polynomial in *x*, $P_n^{(\alpha, \beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1 - x)^\alpha (1 + x)^\beta$.

Examples

```
>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
```

```
>>> 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, Rational(1, 2), Rational(1, 2), x)
RisingFactorial(3/2, n)*chebyshevu(n, x)/factorial(n + 1)
```

```
>>> jacobi(n, -Rational(1, 2), -Rational(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 + b + n + 1)*jacobi(n - 1, a + 1, b + 1, x)/2
```

See also:

[gegenbauer](#) (page 377), [chebyshevt_root](#) (page 380), [chebyshevu](#) (page 379), [chebyshevu_root](#) (page 380), [legendre](#) (page 381), [assoc_legendre](#) (page 381), [hermite](#) (page 382), [laguerre](#) (page 383), [assoc_laguerre](#) (page 384), [diofant.polys.orthopolys.jacobi_poly](#) (page 543), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 543), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 543), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 543), [diofant.polys.orthopolys.hermite_poly](#) (page 543), [diofant.polys.orthopolys.legendre_poly](#) (page 543), [diofant.polys.orthopolys.laguerre_poly](#) (page 543)

References

- https://en.wikipedia.org/wiki/Jacobi_polynomials
- <https://mathworld.wolfram.com/JacobiPolynomial.html>
- <http://functions.wolfram.com/Polynomials/JacobiP/>

`diofant.functions.special.polynomials.jacobi_normalized(n, a, b, x)`

Jacobi polynomial $P_n^{(\alpha, \beta)}(x)$

`jacobi_normalized(n, alpha, beta, x)` gives the n th Jacobi polynomial in x , $P_n^{(\alpha, \beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1-x)^\alpha (1+x)^\beta$.

This functions returns the polynomials normilzed:

$$\int_{-1}^1 P_m^{(\alpha, \beta)}(x) P_n^{(\alpha, \beta)}(x) (1-x)^\alpha (1+x)^\beta dx = \delta_{m,n}$$

Examples

```
>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2**(a + b + 1)*gamma(a + n + 1)*gamma(b + n + 1)/((a + b + 1)*factorial(n)*gamma(a + b + n + 1)))
```

See also:

gegenbauer (page 377), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Jacobi_polynomials
- <https://mathworld.wolfram.com/JacobiPolynomial.html>
- <http://functions.wolfram.com/Polynomials/JacobiP/>

Gegenbauer Polynomials

`class diofant.functions.special.polynomials.gegenbauer(n, a, x)`

Gegenbauer polynomial $C_n^{(\alpha)}(x)$

`gegenbauer(n, alpha, x)` gives the n th Gegenbauer polynomial in x , $C_n^{(\alpha)}(x)$.

The Gegenbauer polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1 - x^2)^{\alpha - \frac{1}{2}}$.

Examples

```
>>> 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)
```

See also:

[*jacobi*](#) (page 374), [*chebyshevt_root*](#) (page 380), [*chebyshevu*](#) (page 379), [*chebyshevu_root*](#) (page 380), [*legendre*](#) (page 381), [*assoc_legendre*](#) (page 381), [*hermite*](#) (page 382), [*laguerre*](#) (page 383), [*assoc_laguerre*](#) (page 384), [*diofant.polys.orthopolys.jacobi_poly*](#) (page 543), [*diofant.polys.orthopolys.gegenbauer_poly*](#) (page 543), [*diofant.polys.orthopolys.chebyshevt_poly*](#) (page 543), [*diofant.polys.orthopolys.chebyshevu_poly*](#) (page 543), [*diofant.polys.orthopolys.hermite_poly*](#) (page 543), [*diofant.polys.orthopolys.legendre_poly*](#) (page 543), [*diofant.polys.orthopolys.laguerre_poly*](#) (page 543)

References

- https://en.wikipedia.org/wiki/Gegenbauer_polynomials
- <https://mathworld.wolfram.com/GegenbauerPolynomial.html>
- <http://functions.wolfram.com/Polynomials/GegenbauerC3/>

Chebyshev Polynomials

class diofant.functions.special.polynomials.chebyshevt(*n*, *x*)

Chebyshev polynomial of the first kind, $T_n(x)$

chebyshevt(*n*, *x*) gives the *n*th Chebyshev polynomial (of the first kind) in *x*, $T_n(x)$.

The Chebyshev polynomials of the first kind are orthogonal on $[-1, 1]$ with respect to the weight $\frac{1}{\sqrt{1-x^2}}$.

Examples

```
>>> 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)
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Chebyshev_polynomial
- <https://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- <https://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- <http://functions.wolfram.com/Polynomials/ChebyshevU/>

class diofant.functions.special.polynomials.chebyshevu(n, x)

Chebyshev polynomial of the second kind, $U_n(x)$

chebyshevu(n, x) gives the n th Chebyshev polynomial of the second kind in x , $U_n(x)$.

The Chebyshev polynomials of the second kind are orthogonal on $[-1, 1]$ with respect to the weight $\sqrt{1 - x^2}$.

Examples

```
>>> 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)
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevt_root* (page 380), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Chebyshev_polynomial
- <https://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- <https://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- <http://functions.wolfram.com/Polynomials/ChebyshevU/>

class diofant.functions.special.polynomials.**chebyshevt_root**(*n*, *k*)

`chebyshevt_root`(*n*, *k*) returns the *k*th root (indexed from zero) of the *n*th Chebyshev polynomial of the first kind; that is, if $0 \leq k < n$, `chebyshevt(n, chebyshevt_root(n, k)) == 0`.

Examples

```
>>> chebyshevt_root(3, 2)
-sqrt(3)/2
>>> chebyshevt(3, chebyshevt_root(3, 2))
0
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

class diofant.functions.special.polynomials.**chebyshevu_root**(*n*, *k*)

`chebyshevu_root`(*n*, *k*) returns the *k*th root (indexed from zero) of the *n*th Chebyshev polynomial of the second kind; that is, if $0 \leq k < n$, `chebyshevu(n, chebyshevu_root(n, k)) == 0`.

Examples

```
>>> chebyshevu_root(3, 2)
-sqrt(2)/2
>>> chebyshevu(3, chebyshevu_root(3, 2))
0
```

See also:

chebyshevt (page 378), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

Legendre Polynomials

class diofant.functions.special.polynomials.**Legendre**(*n*, *x*)

legendre(*n*, *x*) gives the *n*th 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*.

Examples

```
>>> 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)
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Legendre_polynomial
- <https://mathworld.wolfram.com/LegendrePolynomial.html>
- <http://functions.wolfram.com/Polynomials/LegendreP/>
- <http://functions.wolfram.com/Polynomials/LegendreP2/>

class diofant.functions.special.polynomials.**assoc_legendre**(*n*, *m*, *x*)

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 *n*th order Legendre polynomial, $P_n(x)$ in the following manner:

$$P_n^m(x) = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{d^m P_n(x)}{dx^m}$$

Associated Legendre polynomial are orthogonal on $[-1, 1]$ with:

- weight = 1 for the same *m*, and different *n*.
- weight = $1/(1-x^2)$ for the same *n*, and different *m*.

Examples

```
>>> 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)
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Associated_Legendre_polynomials
- <https://mathworld.wolfram.com/LegendrePolynomial.html>
- <http://functions.wolfram.com/Polynomials/LegendreP/>
- <http://functions.wolfram.com/Polynomials/LegendreP2/>

Hermite Polynomials

class `diofant.functions.special.polynomials.hermite`(*n*, *x*)

`hermite`(*n*, *x*) gives the *n*th Hermite polynomial in *x*, $H_n(x)$

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight $\exp(-x^2)$.

Examples

```
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n, x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *laguerre* (page 383), *assoc_laguerre*

(page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Hermite_polynomial
- <https://mathworld.wolfram.com/HermitePolynomial.html>
- <http://functions.wolfram.com/Polynomials/HermiteH/>

Laguerre Polynomials

class `diofant.functions.special.polynomials.laguerre(n, x)`

Returns the n th Laguerre polynomial in x , $L_n(x)$.

Parameters

n (*int*) – Degree of Laguerre polynomial. Must be $n \geq 0$.

Examples

```
>>> 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)
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *assoc_laguerre* (page 384), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- https://en.wikipedia.org/wiki/Laguerre_polynomial
- <https://mathworld.wolfram.com/LaguerrePolynomial.html>
- <http://functions.wolfram.com/Polynomials/LaguerreL/>
- <http://functions.wolfram.com/Polynomials/LaguerreL3/>

class diofant.functions.special.polynomials.assoc_laguerre(*n*, *alpha*, *x*)

Returns the *n*th generalized Laguerre polynomial in *x*, $L_n(x)$.

Parameters

- **n** (*int*) – Degree of Laguerre polynomial. Must be $n \geq 0$.
- **alpha** (*Expr*) – Arbitrary expression. For $\alpha=0$ regular Laguerre polynomials will be generated.

Examples

```
>>> 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))
```

See also:

jacobi (page 374), *gegenbauer* (page 377), *chebyshevt* (page 378), *chebyshevt_root* (page 380), *chebyshevu* (page 379), *chebyshevu_root* (page 380), *legendre* (page 381), *assoc_legendre* (page 381), *hermite* (page 382), *laguerre* (page 383), *diofant.polys.orthopolys.jacobi_poly* (page 543), *diofant.polys.orthopolys.gegenbauer_poly* (page 543), *diofant.polys.orthopolys.chebyshevt_poly* (page 543), *diofant.polys.orthopolys.chebyshevu_poly* (page 543), *diofant.polys.orthopolys.hermite_poly* (page 543), *diofant.polys.orthopolys.legendre_poly* (page 543), *diofant.polys.orthopolys.laguerre_poly* (page 543)

References

- <https://mathworld.wolfram.com/AssociatedLaguerrePolynomial.html>
- <http://functions.wolfram.com/Polynomials/LaguerreL/>
- <http://functions.wolfram.com/Polynomials/LaguerreL3/>

Spherical Harmonics

class diofant.functions.special.spherical_harmonics.Ynm(*n, m, theta, phi*)

Spherical harmonics defined as

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\varphi) P_n^m(\cos(\theta))$$

Ynm() gives the spherical harmonic function of order n and m in θ and φ , $Y_n^m(\theta, \varphi)$. The four parameters are as follows: $n \geq 0$ an integer and m an integer such that $-n \leq m \leq n$ holds. The two angles are real-valued with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$.

Examples

```
>>> theta = Symbol('theta')
>>> phi = Symbol('phi')
```

```
>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)
```

Several symmetries are known, for the order

```
>>> theta = Symbol('theta')
>>> phi = Symbol('phi')
```

```
>>> Ynm(n, -m, theta, phi)
(-1)**m*E**(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

as well as for the angles

```
>>> theta = Symbol('theta')
>>> phi = Symbol('phi')
```

```
>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)
```

```
>>> Ynm(n, m, theta, -phi)
E**(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions

```
>>> simplify(Ynm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
```

```
>>> simplify(Ynm(1, -1, theta, phi).expand(func=True))
sqrt(6)*E**(-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)*E**(I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, -2, theta, phi).expand(func=True))
sqrt(30)*E**(-2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, -1, theta, phi).expand(func=True))
sqrt(30)*E**(-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)*E**(I*phi)*sin(2*theta)/(8*sqrt(pi))
```

```
>>> simplify(Ynm(2, 2, theta, phi).expand(func=True))
sqrt(30)*E**(2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

We can differentiate the functions with respect to both angles

```
>>> theta = Symbol('theta')
>>> phi = Symbol('phi')
```

```
>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + E**(-I*phi)*sqrt((-m + n)*(m + n + 1))*Ynm(n,
↪ m + 1, theta, phi)
```

```
>>> diff(Ynm(n, m, theta, phi), phi)
I*m*Ynm(n, m, theta, phi)
```

Further we can compute the complex conjugation

```
>>> theta = Symbol('theta')
>>> phi = Symbol('phi')
>>> m = Symbol('m')
```

```
>>> conjugate(Ynm(n, m, theta, phi))
(-1)**(2*m)*E**(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

To get back the well known expressions in spherical coordinates we use full expansion

```
>>> theta = Symbol('theta')
>>> phi = Symbol('phi')
```

```
>>> expand_func(Ynm(n, m, theta, phi))
E**(I*m*phi)*sqrt((2*n + 1)*factorial(-m + n)/factorial(m + n))*assoc_legendre(n,
↪ m, cos(theta))/(2*sqrt(pi))
```

See also:

[*diofant.functions.special.spherical_harmonics.Ynm_c*](#) (page 387), [*diofant.functions.special.spherical_harmonics.Znm*](#) (page 387)

References

- https://en.wikipedia.org/wiki/Spherical_harmonics
- <https://mathworld.wolfram.com/SphericalHarmonic.html>
- <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- <https://dlmf.nist.gov/14.30>

`diofant.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:

`diofant.functions.special.spherical_harmonics.Ynm` (page 385), `diofant.functions.special.spherical_harmonics.Znm` (page 387)

References

- https://en.wikipedia.org/wiki/Spherical_harmonics
- <https://mathworld.wolfram.com/SphericalHarmonic.html>
- <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>

class `diofant.functions.special.spherical_harmonics.Znm(n, m, theta, phi)`

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:

`diofant.functions.special.spherical_harmonics.Ynm` (page 385), `diofant.functions.special.spherical_harmonics.Ynm_c` (page 387)

References

- https://en.wikipedia.org/wiki/Spherical_harmonics
- <https://mathworld.wolfram.com/SphericalHarmonic.html>
- <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>

Tensor Functions

`diofant.functions.special.tensor_functions.Eijk(*args, **kwargs)`

Represent the Levi-Civita symbol.

This is just compatibility wrapper to `LeviCivita()`.

See also:

`diofant.functions.special.tensor_functions.LeviCivita` (page 388)

`diofant.functions.special.tensor_functions.eval_levicivita(*args)`

Evaluate Levi-Civita symbol.

class `diofant.functions.special.tensor_functions.LeviCivita(*args)`

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.

Examples

```
>>> from diofant.abc import i, j
>>> 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
```

See also:

`diofant.functions.special.tensor_functions.Eijk` (page 388)

class `diofant.functions.special.tensor_functions.KroneckerDelta(i, j)`

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.

Examples

A simple example with integer indices:

```
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from diofant.abc import i, j
>>> 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)
```

See also:

diofant.functions.special.tensor_functions.KroneckerDelta.eval (page 389),
diofant.functions.special.delta_functions.DiracDelta (page 318)

References

- https://en.wikipedia.org/wiki/Kronecker_delta

classmethod `eval(i, j)`

Evaluates the discrete delta function.

Examples

```
>>> from diofant.abc import i, j
```

```
>>> 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)
```

property `indices_contain_equal_information`

Returns True if indices are either both above or below fermi.

Examples

```
>>> 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
```

property `is_above_fermi`

True if Delta can be non-zero above fermi

Examples

```
>>> 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
```

See also:

[*diofant.functions.special.tensor_functions.KroneckerDelta.is_below_fermi*](#) (page 390), [*diofant.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi*](#) (page 391), [*diofant.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi*](#) (page 390)

property `is_below_fermi`

True if Delta can be non-zero below fermi

Examples

```
>>> 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
```

See also:

[*diofant.functions.special.tensor_functions.KroneckerDelta.is_above_fermi*](#) (page 390), [*diofant.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi*](#) (page 390), [*diofant.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi*](#) (page 391)

property `is_only_above_fermi`

True if Delta is restricted to above fermi

Examples

```

>>> 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

```

See also:

diofant.functions.special.tensor_functions.KroneckerDelta.is_above_fermi (page 390), *diofant.functions.special.tensor_functions.KroneckerDelta.is_below_fermi* (page 390), *diofant.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi* (page 391)

property `is_only_below_fermi`

True if Delta is restricted to below fermi

Examples

```

>>> 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

```

See also:

diofant.functions.special.tensor_functions.KroneckerDelta.is_above_fermi (page 390), *diofant.functions.special.tensor_functions.KroneckerDelta.is_below_fermi* (page 390), *diofant.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi* (page 390)

property `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'.

Examples

```

>>> 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

```

See also:

[`diofant.functions.special.tensor_functions.KroneckerDelta.preferred_index`](#) (page 392)

property `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'.

Examples

```
>>> 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
```

See also:

[`diofant.functions.special.tensor_functions.KroneckerDelta.killable_index`](#) (page 391)

4.7 Integrals

The `integrals` module in Diofant implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is [`integrate\(\)`](#) (page 399)

- `integrate(f, x)` returns the indefinite integral $\int f \, dx$
- `integrate(f, (x, a, b))` returns the definite integral $\int_a^b f \, dx$

4.7.1 Examples

Diofant can integrate a vast array of functions. It can integrate polynomial functions:

```
>>> init_printing(pretty_print=True, wrap_line=False, no_global=True)
>>> integrate(x**2 + x + 1, x)

$$\frac{x^3}{3} + \frac{x^2}{2} + x$$

```

Rational functions:

```
>>> integrate(x/(x**2+2*x+1), x)

$$\log(x + 1) + \frac{1}{x + 1}$$

```

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, Diofant will deal with these integrals.

```
>>> integrate(x**2 * exp(x) * cos(x), x)

$$\frac{e^x \cdot x^2 \cdot \sin(x)}{2} + \frac{e^x \cdot x^2 \cdot \cos(x)}{2} - e^x \cdot x \cdot \sin(x) + \frac{e^x \cdot \sin(x)}{2} - \frac{e^x \cdot \cos(x)}{2}$$

```

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

```
>>> integrate(exp(-x**2)*erf(x), x)

$$\frac{\sqrt{\pi} \cdot \operatorname{erf}^2(x)}{4}$$

```

4.7.2 Integral Transforms

Diofant has special support for definite integrals, and integral transforms.

`diofant.integrals.transforms.mellin_transform(f, x, s, **hints)`

Compute the Mellin transform $F(s)$ of $f(x)$,

$$F(s) = \int_0^{\infty} x^{s-1} f(x) dx.$$

For all “sensible” functions, this converges absolutely in a strip

$$a < \operatorname{Re}(s) < b.$$

The Mellin transform is related via change of variables to the Fourier transform, and also to the (bilateral) Laplace transform.

This function returns $(F, (a, b), \text{cond})$ where F is the Mellin transform of f , (a, b) is the fundamental strip (as above), and cond are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated *MellinTransform* (page 409) object.

For a description of possible hints, refer to the docstring of `diofant.integrals.transforms.IntegralTransform.doit()` (page 409). If `noconds=False`, then only F will be returned (i.e. not cond , and also not the strip (a, b)).

```
>>> from diofant.abc import s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)
```

See also:

inverse_mellin_transform (page 393), *laplace_transform* (page 394), *fourier_transform* (page 395), *hankel_transform* (page 397), *inverse_hankel_transform* (page 398)

`diofant.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

$$f(x) = \int_{c-i\infty}^{c+i\infty} x^{-s} F(s) ds,$$

for any c in the fundamental strip. Under certain regularity conditions on F and/or f , this recovers f from its Mellin transform F (and vice versa), for positive real x .

One of a or b may be passed as `None`; a suitable c will be inferred.

If the integral cannot be computed in closed form, this function returns an unevaluated [InverseMellinTransform](#) (page 409) object.

Note that this function will assume x to be positive and real, regardless of the diofant assumptions!

For a description of possible hints, refer to the docstring of [diofant.integrals.transforms.IntegralTransform.doit\(\)](#) (page 409).

```
>>> from diofant.abc import s
>>> inverse_mellin_transform(gamma(s), s, x, (0, oo))
E**(-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 393), [hankel_transform](#) (page 397), [inverse_hankel_transform](#) (page 398)

`diofant.integrals.transforms.laplace_transform($f, t, s, **hints$)`

Compute the Laplace Transform $F(s)$ of $f(t)$,

$$F(s) = \int_0^{\infty} e^{-st} f(t) dt.$$

For all “sensible” functions, this converges absolutely in a half plane $a < \operatorname{Re}(s)$.

This function returns (F, a, cond) where F is the Laplace transform of f , $\operatorname{Re}(s) > a$ is the half-plane of convergence, and cond are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated [LaplaceTransform](#) (page 409) object.

For a description of possible hints, refer to the docstring of [diofant.integrals.transforms.IntegralTransform.doit\(\)](#) (page 409). If `noconds=True`, only F will be returned (i.e. not cond , and also not the plane a).

```
>>> from diofant.abc import s
>>> laplace_transform(t**a, t, s)
(s**(-a)*gamma(a + 1)/s, 0, -re(a) < 1)
```

See also:

[inverse_laplace_transform](#) (page 394), [mellin_transform](#) (page 393), [fourier_transform](#) (page 395), [hankel_transform](#) (page 397), [inverse_hankel_transform](#) (page 398)

`diofant.integrals.transforms.inverse_laplace_transform($F, s, t, \text{plane}=\text{None}, **hints$)`

Compute the inverse Laplace transform of $F(s)$, defined as

$$f(t) = \int_{c-i\infty}^{c+i\infty} e^{st} F(s) ds,$$

for c so large that $F(s)$ has no singularities in the half-plane $\operatorname{Re}(s) > c - \epsilon$.

The plane can be specified by argument `plane`, but will be inferred if passed as `None`.

Under certain regularity conditions, this recovers $f(t)$ from its Laplace Transform $F(s)$, for non-negative t , and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated [`InverseLaplaceTransform`](#) (page 409) object.

Note that this function will always assume t to be real, regardless of the diofant assumption on t .

For a description of possible hints, refer to the docstring of [`diofant.integrals.transforms.IntegralTransform.doit\(\)`](#) (page 409).

```
>>> from diofant.abc import s
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

See also:

[`laplace_transform`](#) (page 394), [`hankel_transform`](#) (page 397), [`inverse_hankel_transform`](#) (page 398)

`diofant.integrals.transforms.fourier_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency Fourier transform of f , defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [`FourierTransform`](#) (page 410) object.

For a description of possible hints, refer to the docstring of [`diofant.integrals.transforms.IntegralTransform.doit\(\)`](#) (page 409). Note that for this transform, by default `noconds=True`.

```
>>> fourier_transform(exp(-x**2), x, k)
E**(-pi**2*k**2)*sqrt(pi)
>>> fourier_transform(exp(-x**2), x, k, noconds=False)
(E**(-pi**2*k**2)*sqrt(pi), True)
```

See also:

[`inverse_fourier_transform`](#) (page 395), [`sine_transform`](#) (page 396), [`inverse_sine_transform`](#) (page 396), [`cosine_transform`](#) (page 397), [`inverse_cosine_transform`](#) (page 397), [`hankel_transform`](#) (page 397), [`inverse_hankel_transform`](#) (page 398), [`mellin_transform`](#) (page 393), [`laplace_transform`](#) (page 394)

`diofant.integrals.transforms.inverse_fourier_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse Fourier transform of F , defined as

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i x k} dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseFourierTransform* (page 410) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 409). Note that for this transform, by default `noconds=True`.

```
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x)
E**(-x**2)
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x, noconds=False)
(E**(-x**2), True)
```

See also:

fourier_transform (page 395), *sine_transform* (page 396), *inverse_sine_transform* (page 396), *cosine_transform* (page 397), *inverse_cosine_transform* (page 397), *hankel_transform* (page 397), *inverse_hankel_transform* (page 398), *mellin_transform* (page 393), *laplace_transform* (page 394)

`diofant.integrals.transforms.sine_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency sine transform of f , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \sin(2\pi x k) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *SineTransform* (page 410) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 409). Note that for this transform, by default `noconds=True`.

```
>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*E**((-k**2/(4*a))*k/(4*sqrt(a)**3)
>>> 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 395), *inverse_fourier_transform* (page 395), *inverse_sine_transform* (page 396), *cosine_transform* (page 397), *inverse_cosine_transform* (page 397), *hankel_transform* (page 397), *inverse_hankel_transform* (page 398), *mellin_transform* (page 393), *laplace_transform* (page 394)

`diofant.integrals.transforms.inverse_sine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse sine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \sin(2\pi x k) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseSineTransform* (page 410) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 409). Note that for this transform, by default `noconds=True`.

```
>>> 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)
E**(-a*x**2)*x
```


See also:

[fourier_transform](#) (page 395), [inverse_fourier_transform](#) (page 395),
[sine_transform](#) (page 396), [cosine_transform](#) (page 397), [in-](#)
[verse_cosine_transform](#) (page 397), [hankel_transform](#) (page 397), [in-](#)
[verse_hankel_transform](#) (page 398), [mellin_transform](#) (page 393),
[laplace_transform](#) (page 394)

`diofant.integrals.transforms.cosine_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency cosine transform of f , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \cos(2\pi x k) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [CosineTransform](#) (page 410) object.

For a description of possible hints, refer to the docstring of [diofant.integrals.transforms.IntegralTransform.doit\(\)](#) (page 409). Note that for this transform, by default `noconds=True`.

```
>>> 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)
E**(-a**2/(2*k))*a/(2*sqrt(k)**3)
```

See also:

[fourier_transform](#) (page 395), [inverse_fourier_transform](#) (page 395),
[sine_transform](#) (page 396), [inverse_sine_transform](#) (page 396), [in-](#)
[verse_cosine_transform](#) (page 397), [hankel_transform](#) (page 397), [in-](#)
[verse_hankel_transform](#) (page 398), [mellin_transform](#) (page 393),
[laplace_transform](#) (page 394)

`diofant.integrals.transforms.inverse_cosine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse cosine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \cos(2\pi x k) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseCosineTransform](#) (page 410) object.

For a description of possible hints, refer to the docstring of [diofant.integrals.transforms.IntegralTransform.doit\(\)](#) (page 409). Note that for this transform, by default `noconds=True`.

```
>>> inverse_cosine_transform(sqrt(2)*a/(sqrt(pi)*(a**2 + k**2)), k, x)
E**(-a*x)
>>> inverse_cosine_transform(1/sqrt(k), k, x)
1/sqrt(x)
```

See also:

[fourier_transform](#) (page 395), [inverse_fourier_transform](#) (page 395),
[sine_transform](#) (page 396), [inverse_sine_transform](#) (page 396), [cosine_transform](#)
(page 397), [hankel_transform](#) (page 397), [inverse_hankel_transform](#) (page 398),
[mellin_transform](#) (page 393), [laplace_transform](#) (page 394)

`diofant.integrals.transforms.hankel_transform(f, r, k, nu, **hints)`

Compute the Hankel transform of f , defined as

$$F_\nu(k) = \int_0^\infty f(r) J_\nu(kr) r dr.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *HankelTransform* (page 410) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 409). Note that for this transform, by default `noconds=True`.

```
>>> from diofant.abc import k, nu, r
```

```
>>> 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*sqrt(a**2/k**2 + 1)**3)
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
E**(-a*r)
```

See also:

fourier_transform (page 395), *inverse_fourier_transform* (page 395), *sine_transform* (page 396), *inverse_sine_transform* (page 396), *cosine_transform* (page 397), *inverse_cosine_transform* (page 397), *inverse_hankel_transform* (page 398), *mellin_transform* (page 393), *laplace_transform* (page 394)

`diofant.integrals.transforms.inverse_hankel_transform(F, k, r, nu, **hints)`

Compute the inverse Hankel transform of F defined as

$$f(r) = \int_0^\infty F_\nu(k) J_\nu(kr) k dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseHankelTransform* (page 410) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 409). Note that for this transform, by default `noconds=True`.

```
>>> from diofant.abc import k, nu, r
```

```
>>> 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*sqrt(a**2/k**2 + 1)**3)
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
E**(-a*r)
```

See also:

[fourier_transform](#) (page 395), [inverse_fourier_transform](#) (page 395), [sine_transform](#) (page 396), [inverse_sine_transform](#) (page 396), [cosine_transform](#) (page 397), [inverse_cosine_transform](#) (page 397), [hankel_transform](#) (page 397), [mellin_transform](#) (page 393), [laplace_transform](#) (page 394)

4.7.3 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.

Diofant 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. Diofant also uses pattern matching and heuristics to speed up evaluation of some types of integrals, e.g. polynomials.

For non-elementary definite integrals, Diofant uses so-called Meijer G-functions. Details are described [here](#) (page 798).

4.7.4 API reference

`diofant.integrals.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

Diofant 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.

Diofant 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).

In general, the algebraic methods work best for computing antiderivatives of (possibly complicated) combinations of elementary functions. The G-function methods work best for computing definite integrals from zero to infinity of moderately complicated combinations of special functions, or indefinite integrals of very simple combinations of special functions.

The strategy employed by the integration code is as follows:

- If computing a definite integral, and both limits are real, and at least one limit is $+\infty$, try the G-function method of definite integration first.
- Try to find an antiderivative, using all available methods, ordered by performance (that is try fastest method first, slowest last; in particular polynomial integration is tried first, Meijer G-functions second to last, and heuristic Risch last).
- If still not successful, try G-functions irrespective of the limits.

The option `meijerg=True, False, None` can be used to, respectively: always use G-function methods and no others, never use G-function methods, or use all available methods (in order as described above). It defaults to `None`.

Examples

```
>>> 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:

```
>>> integrate(sqrt(1 + x), (x, 0, x))
2*sqrt(x + 1)**3/3 - 2/3
>>> integrate(sqrt(1 + x), x)
2*sqrt(x + 1)**3/3
```

```
>>> 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(E**(-x)*x**a, (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)
```

See also:

[`diofant.integrals.integrals.Integral`](#) (page 405), [`diofant.integrals.integrals.Integral.doit`](#) (page 406)

`diofant.integrals.deltafunctions.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, then 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.

Examples

```
>>> 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)
```

See also:

[`diofant.functions.special.delta_functions.DiracDelta`](#) (page 318), [`diofant.integrals.integrals.Integral`](#) (page 405)

`diofant.integrals.rationaltools.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 $f = g'$.

```
>>> 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)
```

References

- [Bro05], pp. 35-70

See also:

[`diofant.integrals.integrals.Integral.doit`](#) (page 406), [`ratint_logpart`](#) (page 402), [`ratint_ratpart`](#) (page 403)

`diofant.integrals.rationaltools.ratint_logpart(f, g, x, t=None)`

Lazard-Rioboo-Trager algorithm.

Given a field K and polynomials f and g in $K[x]$, such that f and g are coprime, $\deg(f) < \deg(g)$ and g is square-free, returns a list of tuples (s_i, q_i) of polynomials, for $i = 1..n$, such that s_i in $K[t, x]$ and q_i in $K[t]$, and:

$$\frac{d}{dx} \frac{f}{g} = \frac{d}{dx} \left(\prod_{i=1..n} \frac{s_i}{q_i} \right) = \sum_{i=1..n} \frac{s_i'}{q_i} + \sum_{i=1..n} \frac{s_i}{q_i} \log(s_i(a, x))$$

Examples

```
>>> ratint_logpart(1, x**2 + x + 1, x)
[(Poly(x + 3*_t/2 + 1/2, x, domain='QQ[_t]'),
 Poly(3*_t**2 + 1, _t, domain='ZZ'))]
>>> ratint_logpart(12, x**2 - x - 2, x)
[(Poly(x - 3*_t/8 - 1/2, x, domain='QQ[_t]'),
 Poly(_t**2 - 16, _t, domain='ZZ'))]
```

See also:

[ratint](#) (page 402), [ratint_ratpart](#) (page 403)

`diofant.integrals.rationaltools.ratint_ratpart(f, g, x)`

Horowitz-Ostrogradsky algorithm.

Given a field K and polynomials f and g in $K[x]$, such that f and g are coprime and $\deg(f) < \deg(g)$, returns fractions A and B in $K(x)$, such that $f/g = A' + B$ and B has square-free denominator.

Examples

```
>>> ratint_ratpart(1, x + 1, x)
(0, 1/(x + 1))
>>> ratint_ratpart(1, x**2 + y**2, x)
(0, 1/(x**2 + y**2))
>>> ratint_ratpart(36, x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2, x)
((12*x + 6)/(x**2 - 1), 12/(x**2 - x - 2))
```

See also:

[ratint](#) (page 402), [ratint_logpart](#) (page 402)

`diofant.integrals.heurisch.components(f, x)`

Returns a set of all functional components of the given expression which includes symbols, function applications and compositions and non-integer powers. Fractional powers are collected with minimal, positive exponents.

```
>>> components(sin(x)*cos(x)**2, x)
{x, sin(x), cos(x)}
```

See also:

[heurisch](#) (page 403)

`diofant.integrals.heurisch.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 function 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.

Parameters

- **f** (*Expr*) - expression
- **x** (*Symbol*) - variable
- **rewrite** (*Boolean, optional*) - force rewrite 'f' in terms of 'tan' and 'tanh', default False.
- **hints** (*None or list*) - a list of functions that may appear in anti-derivate. If None (default) - no suggestions at all, if empty list - try to figure out.

Examples

```
>>> heurisch(y*tan(x), x)
y*log(tan(x)**2 + 1)/2
```

References

- [\[Bro\]](#)

See also:

[diofant.integrals.integrals.Integral.doit](#) (page 406), [diofant.integrals.integrals.Integral](#) (page 405), [components](#) (page 403)

`diofant.integrals.heurisch.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.

Examples

```
>>> heurisch(cos(n*x), x)
sin(n*x)/n
>>> heurisch_wrapper(cos(n*x), x)
Piecewise((x, Eq(n, 0)), (sin(n*x)/n, true))
```

See also:

[heurisch](#) (page 403)

`diofant.integrals.trigonometry.trigintegrate(f, x, conds='piecewise')`

Integrate $f = \text{Mul}(\text{trig})$ over 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)
```

References

- https://en.wikibooks.org/wiki/Calculus/Integration_techniques

See also:

`diofant.integrals.integrals.Integral.doit` (page 406), `diofant.integrals.integrals.Integral` (page 405)

The class *Integral* represents an unevaluated integral and has some methods that help in the integration of an expression.

class `diofant.integrals.integrals.Integral(function, *symbols, **assumptions)`

Represents an 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.

References

- https://en.wikipedia.org/wiki/Rectangle_method

Examples

```
>>> 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).evalf(4)
1.730
>>> e.as_sum(10).evalf(4)
1.809
>>> e.doit().evalf(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
```

See also:

[diofant.integrals.integrals.Integral.doit](#) (page 406)

Perform the integration using any hints

`doit(hints)`**

Perform the integration using any hints given.

Examples

```
>>> from diofant.abc import i
>>> Integral(x**i, (i, 1, 3)).doit()
Piecewise((2, Eq(log(x), 0)), (x**3/log(x) - x/log(x), true))
```

See also:

[diofant.integrals.trigonometry.trigintegrate](#) (page 404), [diofant.integrals.heurisch.heurisch](#) (page 403), [diofant.integrals.rationaltools.ratint](#) (page 402)

[diofant.integrals.integrals.Integral.as_sum](#) (page 405)

Approximate the integral using a sum

property 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.

Examples

```
>>> Integral(x, (x, y, 1)).free_symbols
{y}
```

See also:

[*diofant.concrete.expr_with_limits.ExprWithLimits.function*](#) (page 267),
[*diofant.concrete.expr_with_limits.ExprWithLimits.limits*](#) (page 268),
[*diofant.concrete.expr_with_limits.ExprWithLimits.variables*](#) (page 268)

transform(*x*, *u*)

Performs a change of variables from x to u using the relationship given by x and u which will define the transformations f and F (which are inverses of each other) as follows:

- 1) If x is a Symbol (which is a variable of integration) then u will be interpreted as some function, $f(u)$, with inverse $F(u)$. This, in effect, just makes the substitution of x with $f(x)$.
- 2) If u is a Symbol then x will be interpreted as some function, $F(x)$, with inverse $f(u)$. This is commonly referred to as u -substitution.

Once f and F have been identified, the transformation is made as follows:

$$\int_a^b x dx \rightarrow \int_{F(a)}^{F(b)} f(x) \frac{d}{dx}$$

where $F(x)$ is the inverse of $f(x)$ and the limits and integrand have been corrected so as to retain the same value after integration.

Notes

The mappings, $F(x)$ or $f(u)$, must lead to a unique integral. Linear or rational linear expression, $2*x$, $1/x$ and $\text{sqrt}(x)$, will always work; quadratic expressions like $x**2-1$ are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if x is not a variable of integration.

x must be (or contain) only one of the integration variables. If u has more than one free symbol then it should be sent as a tuple $(u, uvar)$ where $uvar$ identifies which variable is replacing the integration variable. XXX can it contain another integration variable?

Examples

```
>>> from diofant.abc import u
```

```
>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))
```

transform can change the variable of integration

```
>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))
```

transform can perform u-substitution as long as a unique integrand is obtained:

```
>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```

This attempt fails because $x = \pm\sqrt{u + 1}$ and the sign does not cancel out of the integrand:

```
>>> Integral(cos(x**2 - 1), (x, 0, 1)).transform(x**2 - 1, u)
Traceback (most recent call last):
ValueError:
The mapping between F(x) and f(u) did not give a unique integrand.
```

transform can do a substitution. Here, the previous result is transformed back into the original expression using “u-substitution”:

```
>>> ui =
>>> _transform(sqrt(u + 1), x) == i
True
```

We can accomplish the same with a regular substitution:

```
>>> ui.transform(u, x**2 - 1) == i
True
```

If the x does not contain a symbol of integration then the integral will be returned unchanged. Integral i does not have an integration variable a so no change is made:

```
>>> i.transform(a, x) == i
True
```

When u has more than one free symbol the symbol that is replacing x must be identified by passing u as a tuple:

```
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, u))
Integral(a + u, (u, -a, -a + 1))
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, a))
Integral(a + u, (a, -u, -u + 1))
```

See also:

[*diofant.concrete.expr_with_limits.ExprWithLimits.variables*](#) (page 268)

Lists the integration variables

[*diofant.concrete.expr_with_limits.ExprWithLimits.as_dummy*](#) (page 267)

Replace integration variables with dummy ones

class `diofant.integrals.transforms.IntegralTransform(*args)`

Base class for integral transforms.

This class represents unevaluated transforms.

To implement a concrete transform, derive from this class and implement the `_compute_transform(f, x, s, **hints)` and `_as_integral(f, x, s)` functions. If the transform cannot be computed, raise `IntegralTransformError`.

Also set `cls._name`.

Implement `self.collapse_extra` if your function returns more than just a number and possibly a convergence condition.

doit(hints)**

Try to evaluate the transform in closed form.

This general function handles linearity, but apart from that leaves pretty much everything to `_compute_transform`.

Standard hints are the following:

- `simplify`: whether or not to simplify the result
- `noconds`: if True, don't return convergence conditions
- **needeval**: if True, raise `IntegralTransformError` instead of returning `IntegralTransform` objects

The default values of these hints depend on the concrete transform, usually the default is `(simplify, noconds, needeval) = (True, False, False)`.

property free_symbols

This method returns the symbols that will exist when the transform is evaluated.

property function

The function to be transformed.

property function_variable

The dependent variable of the function to be transformed.

property transform_variable

The independent transform variable.

class diofant.integrals.transforms.**MellinTransform**(*args)

Class representing unevaluated Mellin transforms.

See also:

[*IntegralTransform*](#) (page 408), [*mellin_transform*](#) (page 393)

class diofant.integrals.transforms.**InverseMellinTransform**(*args)

Class representing unevaluated inverse Mellin transforms.

See also:

[*IntegralTransform*](#) (page 408), [*inverse_mellin_transform*](#) (page 393)

class diofant.integrals.transforms.**LaplaceTransform**(*args)

Class representing unevaluated Laplace transforms.

See also:

[*IntegralTransform*](#) (page 408), [*laplace_transform*](#) (page 394)

class diofant.integrals.transforms.**InverseLaplaceTransform**(*args)

Class representing unevaluated inverse Laplace transforms.

See also:

[*IntegralTransform*](#) (page 408), [*inverse_laplace_transform*](#) (page 394)

class diofant.integrals.transforms.**FourierTransform**(*args)

Class representing unevaluated Fourier transforms.

See also:

[*IntegralTransform*](#) (page 408), [*fourier_transform*](#) (page 395)

class diofant.integrals.transforms.**InverseFourierTransform**(*args)

Class representing unevaluated inverse Fourier transforms.

See also:

[*IntegralTransform*](#) (page 408), [*inverse_fourier_transform*](#) (page 395)

class diofant.integrals.transforms.**SineTransform**(*args)

Class representing unevaluated sine transforms.

See also:

[*IntegralTransform*](#) (page 408), [*sine_transform*](#) (page 396)

class diofant.integrals.transforms.**InverseSineTransform**(*args)

Class representing unevaluated inverse sine transforms.

See also:

[*IntegralTransform*](#) (page 408), [*inverse_sine_transform*](#) (page 396)

class diofant.integrals.transforms.**CosineTransform**(*args)

Class representing unevaluated cosine transforms.

See also:

[*IntegralTransform*](#) (page 408), [*cosine_transform*](#) (page 397)

class diofant.integrals.transforms.**InverseCosineTransform**(*args)

Class representing unevaluated inverse cosine transforms.

See also:

[*IntegralTransform*](#) (page 408), [*inverse_cosine_transform*](#) (page 397)

class diofant.integrals.transforms.**HankelTransform**(*args)

Class representing unevaluated Hankel transforms.

See also:

[*IntegralTransform*](#) (page 408), [*hankel_transform*](#) (page 397)

class diofant.integrals.transforms.**InverseHankelTransform**(*args)

Class representing unevaluated inverse Hankel transforms.

See also:

[*IntegralTransform*](#) (page 408), [*inverse_hankel_transform*](#) (page 398)

4.7.5 Numeric Integrals

Diofant has functions to calculate points and weights for Gaussian quadrature of any order and any precision:

`diofant.integrals.quadrature.gauss_legendre(n, n_digits)`

Computes the Gauss-Legendre quadrature points and weights.

The Gauss-Legendre quadrature approximates the integral:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of P_n and the weights w_i are given by:

$$w_i = \frac{2}{(1 - x_i^2) (P'_n(x_i))^2}$$

Parameters

- **n** (*the order of quadrature*)
- **n_digits** (*number of significant digits of the points and weights to return*)

Returns

(**x**, **w**) (the x and w are lists of points and weights as Floats.) – The points x_i and weights w_i are returned as (x, w) tuple of lists.

Examples

```
>>> x, w = gauss_legendre(3, 5)
>>> x
[-0.7746, 0, 0.7746]
>>> w
[0.55556, 0.88889, 0.55556]
```

```
>>> x, w = gauss_legendre(4, 5)
>>> x
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> w
[0.34785, 0.65215, 0.65215, 0.34785]
```

See also:

[gauss_laguerre](#) (page 411), [gauss_gen_laguerre](#) (page 413), [gauss_hermite](#) (page 412), [gauss_chebyshev_t](#) (page 414), [gauss_chebyshev_u](#) (page 415), [gauss_jacobi](#) (page 416)

References

- https://en.wikipedia.org/wiki/Gaussian_quadrature

`diofant.integrals.quadrature.gauss_laguerre(n, n_digits)`

Computes the Gauss-Laguerre quadrature points and weights.

The Gauss-Laguerre quadrature approximates the integral:

$$\int_0^\infty e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n and the weights w_i are given by:

$$w_i = \frac{x_i}{(n+1)^2 (L_{n+1}(x_i))^2}$$

Parameters

- **n** (*the order of quadrature*)
- **n_digits** (*number of significant digits of the points and weights to return*)

Returns

(x, w) (the x and w are lists of points and weights as Floats.) - The points x_i and weights w_i are returned as (x, w) tuple of lists.

Examples

```
>>> 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]
```

See also:

[gauss_legendre](#) (page 411), [gauss_gen_laguerre](#) (page 413), [gauss_hermite](#) (page 412), [gauss_chebyshev_t](#) (page 414), [gauss_chebyshev_u](#) (page 415), [gauss_jacobi](#) (page 416)

References

- https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature

diofant.integrals.quadrature.**gauss_hermite**(n, n_digits)

Computes the Gauss-Hermite quadrature points and weights.

The Gauss-Hermite quadrature approximates the integral:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of H_n and the weights w_i are given by:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 (H_{n-1}(x_i))^2}$$

Parameters

- **n** (*the order of quadrature*)
- **n_digits** (*number of significant digits of the points and weights to return*)

Returns

(x, w) (the x and w are lists of points and weights as Floats.) – The points x_i and weights w_i are returned as (x, w) tuple of lists.

Examples

```
>>> 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]
```

See also:

[gauss_legendre](#) (page 411), [gauss_laguerre](#) (page 411), [gauss_gen_laguerre](#) (page 413), [gauss_chebyshev_t](#) (page 414), [gauss_chebyshev_u](#) (page 415), [gauss_jacobi](#) (page 416)

References

- https://en.wikipedia.org/wiki/Gauss-Hermite_Quadrature

diofant.integrals.quadrature.**gauss_gen_laguerre**(n, alpha, n_digits)

Computes the generalized Gauss-Laguerre quadrature 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.

Examples

```
>>> x, w = gauss_gen_laguerre(3, -0.5, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> w
[1.4493, 0.31413, 0.00906]
```

```
>>> x, w = gauss_gen_laguerre(4, 1.5, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

See also:

gauss_legendre (page 411), *gauss_laguerre* (page 411), *gauss_hermite* (page 412), *gauss_chebyshev_t* (page 414), *gauss_chebyshev_u* (page 415), *gauss_jacobi* (page 416)

References

- https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature

`diofant.integrals.quadrature.gauss_chebyshev_t(n, n_digits)`

Computes the Gauss-Chebyshev quadrature 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.

Examples

```
>>> 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]
```

See also:

[gauss_legendre](#) (page 411), [gauss_laguerre](#) (page 411), [gauss_hermite](#) (page 412), [gauss_gen_laguerre](#) (page 413), [gauss_chebyshev_u](#) (page 415), [gauss_jacobi](#) (page 416)

References

- https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature

diofant.integrals.quadrature.**gauss_chebyshev_u**(*n*, *n_digits*)

Computes the Gauss-Chebyshev quadrature 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.

Examples

```
>>> 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]
```

See also:

[gauss_legendre](#) (page 411), [gauss_laguerre](#) (page 411), [gauss_hermite](#) (page 412), [gauss_gen_laguerre](#) (page 413), [gauss_chebyshev_t](#) (page 414), [gauss_jacobi](#) (page 416)

References

- https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature

`diofant.integrals.quadrature.gauss_jacobi(n, alpha, beta, n_digits)`

Computes the Gauss-Jacobi quadrature 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.

Examples

```
>>> x, w = gauss_jacobi(3, 0.5, -0.5, 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]
```

See also:

[gauss_legendre](#) (page 411), [gauss_laguerre](#) (page 411), [gauss_hermite](#) (page 412), [gauss_gen_laguerre](#) (page 413), [gauss_chebyshev_t](#) (page 414), [gauss_chebyshev_u](#) (page 415)

References

- https://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature

4.8 Logic

4.8.1 Introduction

The logic module for Diofant allows to form and manipulate logic expressions using symbolic and Boolean values.

4.8.2 Forming logical expressions

You can build Boolean expressions with the standard python operators & (*And* (page 419)), | (*Or* (page 419)), ~ (*Not* (page 420)):

```
>>> y | (x & y)
y | (x & y)
>>> x | y
x | y
>>> ~x
~x
```

You can also form implications with >> and <<:

```
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in Diofant, Boolean expressions inherit from *Basic* (page 46):

```
>>> (y & x).subs({x: True, y: True})
true
>>> (x | y).atoms()
{x, y}
```

4.8.3 Boolean functions

class diofant.logic.boolalg.**BooleanTrue**(*args, **kwargs)

Diofant version of True, a singleton that can be accessed via `true`.

This is the Diofant version of True, for use in the logic module. The primary advantage of using `true` instead of `True` is that shorthand boolean operations like `~` and `>>` will work as expected on this class, whereas with `True` they act bitwise on 1. Functions in the logic module will return this class when they evaluate to true.

Notes

There is liable to be some confusion as to when `True` should be used and when `true` should be used in various contexts throughout Diofant. An important thing to remember is that `sympify(True)` returns `true`. This means that for the most part, you can just use `True` and it will automatically be converted to `true` when necessary, similar to how you can generally use `1` instead of `Integer(1)`.

The rule of thumb is:

“If the boolean in question can be replaced by an arbitrary symbolic `Boolean`, like `Or(x, y)` or `x > 1`, use `true`. Otherwise, use `True`”.

In other words, use `true` only on those contexts where the boolean is being used as a symbolic representation of truth. For example, if the object ends up in the `.args` of any expression, then it must necessarily be `true` instead of `True`, as elements of `.args` must be `Basic`. On the other hand, `==` is not a symbolic operation in Diofant, since it always returns `True` or `False`, and does so in terms of structural equality rather than mathematical, so it should return `True`. The assumptions system should use `True` and `False`. Aside from not satisfying the above rule of thumb, the assumptions system uses a three-valued logic (`True`, `False`, `None`), whereas `true` and `false` represent a two-valued logic. When in doubt, use `True`.

“`true == True` is `True`.”

While “`true is True`” is `False`, “`true == True`” is `True`, so if there is any doubt over whether a function or expression will return `true` or `True`, just use `==` instead of `is` to do the comparison, and it will work in either case. Finally, for boolean flags, it’s better to just use `if x` instead of `if x is True`. To quote PEP 8:

Don’t compare boolean values to `True` or `False` using `==`.

- Yes: `if greeting:`
- No: `if greeting == True:`
- Worse: `if greeting is True:`

Examples

```
>>> sympify(True)
true
>>> ~true
false
>>> not True
False
>>> Or(True, False)
true
```

See also:

[*BooleanFalse*](#) (page 418)

class diofant.logic.boolalg.**BooleanFalse**(*args, **kwargs)

Diofant version of `False`, a singleton that can be accessed via `false`.

This is the Diofant version of `False`, for use in the logic module. The primary advantage of using `false` instead of `False` is that shorthand boolean operations like `~` and `>>` will work as expected on this class, whereas with `False` they act bitwise on `0`. Functions in the logic module will return this class when they evaluate to `false`.

Notes

See note in [BooleanTrue](#) (page 417).

Examples

```
>>> sympify(False)
false
>>> false >> false
true
>>> False >> False
0
>>> Or(True, False)
true
```

See also:

[BooleanTrue](#) (page 417)

class diofant.logic.boolalg.And(*args)

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.

Examples

```
>>> x & y
x & y
```

Notes

The & operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, And(a, b) and a & b will return different things if a and b are integers.

```
>>> (x & y).subs({x: 1})
y
```

class diofant.logic.boolalg.Or(*args)

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.

Examples

```
>>> x | y
x | y
```

Notes

The `|` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, `Or(a, b)` and `a | b` will return different things if `a` and `b` are integers.

```
>>> (x | y).subs({x: 0})
y
```

class diofant.logic.boolalg.**Not**(arg)

Logical Not function (negation).

Returns True if the statement is False. Returns False if the statement is True.

Examples

```
>>> Not(True)
false
>>> Not(False)
true
>>> ~And(True, False)
true
>>> ~Or(True, False)
false
>>> ~(And(True, x) & Or(x, False))
~x
>>> ~x
~x
>>> ~((x | y) & (~x | ~y))
~((x | y) & (~x | ~y))
```

```
>>> not True
False
>>> ~true
false
```

class diofant.logic.boolalg.**Xor**(*args)

Logical XOR (exclusive OR) function.

Returns True if an odd number of the arguments are True and the rest are False.

Returns False if an even number of the arguments are True and the rest are False.

Examples

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


Notes

The `^` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, `a ^ b` and `Xor(a, b)` will be different if `a` and `b` are integers.

```
>>> (x ^ y).subs({y: 0})
x
```

class diofant.logic.boolalg.**Nand**(*args)

Logical NAND function.

It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False. Returns False if all arguments are True.

Examples

```
>>> Nand(False, True)
true
>>> Nand(True, True)
false
>>> Nand(x, y)
~(x & y)
```

class diofant.logic.boolalg.**Nor**(*args)

Logical NOR function.

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True. Returns True if all arguments are False.

Examples

```
>>> Nor(True, False)
false
>>> Nor(True, True)
false
>>> Nor(False, True)
false
>>> Nor(False, False)
true
>>> Nor(x, y)
~(x | y)
```

class diofant.logic.boolalg.**Implies**(*args)

Logical implication.

A implies B is equivalent to $\neg A \vee B$

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False. Returns True otherwise.

Examples

```
>>> Implies(True, False)
false
>>> Implies(False, False)
true
>>> Implies(True, True)
true
>>> Implies(False, True)
true
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)
```

Notes

The `>>` and `<<` operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, `Implies(a, b)` and `a >> b` will return different things if `a` and `b` are integers. In particular, since Python considers `True` and `False` to be integers, `True >> True` will be the same as `1 >> 1`, i.e., 0, which has a truth value of `False`. To avoid this issue, use the Diofant objects `true` and `false`.

```
>>> True >> False
1
>>> true >> false
false
```

class diofant.logic.boolalg.**Equivalent**(*args)

Equivalence relation.

`Equivalent(A, B)` is `True` iff `A` and `B` are both `True` or both `False`.

Returns `True` if all of the arguments are logically equivalent. Returns `False` otherwise.

Examples

```
>>> Equivalent(False, False, False)
true
>>> Equivalent(True, False, False)
false
>>> Equivalent(x, And(x, True))
true
```

class diofant.logic.boolalg.**ITE**(*args)

If then else clause.

`ITE(A, B, C)` evaluates and returns the result of `B` if `A` is `true` else it returns the result of `C`.

Examples

```

>>> ITE(True, False, True)
false
>>> ITE(Or(True, False), And(True, True), Xor(True, True))
true
>>> ITE(x, y, z)
ITE(x, y, z)
>>> ITE(True, x, y)
x
>>> ITE(False, x, y)
y
>>> ITE(x, y, y)
y

```

The following functions can be used to handle Conjunctive and Disjunctive Normal forms

`diofant.logic.boolalg.to_cnf(expr, simplify=False)`

Convert `expr` to Conjunctive Normal Form (CNF).

If `simplify` is `True`, the `expr` is evaluated to its simplest CNF form.

Examples

```

>>> to_cnf(~(a | b) | c)
(c | ~a) & (c | ~b)
>>> to_cnf((a | b) & (a | ~a), True)
a | b

```

See also:

[`is_cnf`](#) (page 423)

`diofant.logic.boolalg.to_dnf(expr, simplify=False)`

Convert `expr` to Disjunctive Normal Form (DNF).

If `simplify` is `True`, the `expr` is evaluated to its simplest DNF form.

Examples

```

>>> to_dnf(b & (a | c))
(a & b) | (b & c)
>>> to_dnf((a & b) | (a & ~b) | (b & c) | (~b & c), True)
a | c

```

See also:

[`is_dnf`](#) (page 424)

`diofant.logic.boolalg.is_cnf(expr)`

Checks if `expr` is in Conjunctive Normal Form (CNF).

A logical expression is in CNF if it is a conjunction of one or more clauses, where a clause is a disjunction of literals.

Examples

```
>>> is_cnf(a | b | c)
True
>>> is_cnf(a & b & c)
True
>>> is_cnf((a & b) | c)
False
```

See also:

[to_cnf](#) (page 423), [is_dnf](#) (page 424), [is_nnf](#) (page 424)

`diofant.logic.boolalg.is_dnf(expr)`

Checks if `expr` is in Disjunctive Normal Form (DNF).

A logical expression is in DNF if it is a disjunction of one or more clauses, where a clause is a conjunction of literals.

Examples

```
>>> is_dnf(a | b | c)
True
>>> is_dnf(a & b & c)
True
>>> is_dnf((a & b) | c)
True
>>> is_dnf(a & (b | c))
False
```

See also:

[to_dnf](#) (page 423), [is_cnf](#) (page 423), [is_nnf](#) (page 424)

The following functions can be used to handle Negation Normal Forms

`diofant.logic.boolalg.to_nnf(expr, simplify=True)`

Converts `expr` to Negation Normal Form (NNF).

If `simplify` is `True`, the result contains no redundant clauses.

Examples

```
>>> to_nnf(~((~a & ~b) | (c & d)))
(a | b) & (~c | ~d)
>>> to_nnf(Equivalent(a >> b, b >> a))
(a | ~b | (a & ~b)) & (b | ~a | (b & ~a))
```

See also:

[is_nnf](#) (page 424)

`diofant.logic.boolalg.is_nnf(expr, simplified=True)`

Checks if `expr` is in Negation Normal Form (NNF).

A logical expression is in NNF if the negation operator is only applied to literals and the only other allowed boolean functions are conjunction and disjunction.

If `simplified` is `True`, checks if result contains no redundant clauses.

Examples

```
>>> is_nnf(a & b | ~c)
True
>>> is_nnf((a | ~a) & (b | c))
False
>>> is_nnf((a | ~a) & (b | c), False)
True
>>> is_nnf(~(a & b) | c)
False
>>> is_nnf((a >> b) & (b >> a))
False
```

See also:

[to_nnf](#) (page 424), [is_cnf](#) (page 423), [is_dnf](#) (page 424)

4.8.4 Simplification and equivalence-testing

`diofant.logic.boolalg.simplify_logic(expr, form='cnf', deep=True)`

This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an Or or And object in Diofant.

Parameters

- **expr** (*string or boolean expression*)
- **form** (*string ('cnf' or 'dnf'), default to 'cnf'.*) - Selects the normal form in which the result is returned.
- **deep** (*boolean (default True)*) - indicates whether to recursively simplify any non-boolean functions contained within the input.

Examples

```
>>> b = (~x & ~y & ~z) | (~x & ~y & z)
>>> simplify_logic(b)
~x & ~y
```

```
>>> sympify(b)
(z & ~x & ~y) | (~x & ~y & ~z)
>>> simplify_logic(_)
~x & ~y
```

Diofant's `simplify()` function can also be used to simplify logic expressions to their simplest forms.

4.8.5 Inference

This module implements some inference routines in propositional logic.

The function `satisfiable` will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence *True*.

For example, the expression $x \ \& \ \sim x$ is not satisfiable, since there are no values for x that make this sentence *True*. On the other hand, $(x \ | \ y) \ \& \ (x \ | \ \sim y) \ \& \ (\sim x \ | \ y)$ is satisfiable with both x and y being *True*.

```
>>> 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.

`diofant.logic.inference.satisfiable(expr, algorithm='dpll2', all_models=False)`

Check satisfiability of a propositional sentence. Returns a model when it succeeds. Returns {true: true} for trivially true expressions.

On setting `all_models` to True, if given `expr` is satisfiable then returns a generator of models. However, if `expr` is unsatisfiable then returns a generator containing the single element False.

Examples

```
>>> satisfiable(a & ~b)
{a: True, b: False}
>>> satisfiable(a & ~a)
False
>>> satisfiable(True)
{true: True}
>>> next(satisfiable(a & ~a, all_models=True))
False
>>> models = satisfiable((a >> b) & b, all_models=True)
>>> next(models)
{a: False, b: True}
>>> next(models)
{a: True, b: True}
>>> def use_models(models):
...     for model in models:
...         if model:
...             # Do something with the model.
...             return model
...         else:
...             # Given expr is unsatisfiable.
...             print('UNSAT')
>>> use_models(satisfiable(a >> ~a, all_models=True))
{a: False}
>>> use_models(satisfiable(a ^ a, all_models=True))
UNSAT
```

4.9 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 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.

4.9.1 Abstract Domains

class diofant.domains.domain.**Domain**

Represents an abstract domain.

convert(*element*, *base=None*)

Convert element to self.dtype.

convert_from(*element*, *base*)

Convert element to self.dtype given the base domain.

frac_field(**symbols*, ***kwargs*)

Return a fraction field, i.e. $K(X)$.

abstract from_expr(*expr*)

Convert Diofant's expression *expr* to dtype.

get_exact()

Get an associated exact domain.

poly_ring(**symbols*, ***kwargs*)

Return a polynomial ring, i.e. $K[X]$.

abstract to_expr(*element*)

Convert domain element to Diofant expression.

unify(*K1*, *symbols=()*)

Construct a minimal domain that contains elements of self and *K1*.

Known domains (from smallest to largest):

- GF(*p*)
- ZZ
- QQ
- RR(*prec*, *tol*)
- CC(*prec*, *tol*)
- ALG(*a*, *b*, *c*)
- K[*x*, *y*, *z*]
- K(*x*, *y*, *z*)
- EX

class diofant.domains.field.**Field**

Represents a field domain.

div(*a*, *b*)

Division of *a* and *b*, implies `__truediv__`.

exquo(*a*, *b*)

Exact quotient of *a* and *b*, implies `__truediv__`.

property field

Return a field associated with self.

gcd(*a*, *b*)Return GCD of *a* and *b*.This definition of GCD over fields allows to clear denominators in *primitive()*.

```
>>> QQ.gcd(QQ(2, 3), QQ(4, 9))
2/9
>>> gcd(Rational(2, 3), Rational(4, 9))
2/9
>>> primitive(2*x/3 + Rational(4, 9))
(2/9, 3*x + 2)
```

quo(*a*, *b*)Quotient of *a* and *b*, implies `__truediv__`.**rem**(*a*, *b*)Remainder of *a* and *b*, implies nothing.**class** diofant.domains.ring.**CommutativeRing**

Represents a ring domain.

abstract property characteristic

Return the characteristic of this ring.

cofactors(*a*, *b*)Return GCD and cofactors of *a* and *b*.**div**(*a*, *b*)Division of *a* and *b*, implies `__divmod__`.**exquo**(*a*, *b*)Exact quotient of *a* and *b*, implies `__floordiv__`.**half_gcdex**(*a*, *b*)Half extended GCD of *a* and *b*.**invert**(*a*, *b*)Return inversion of *a* mod *b*.**is_normal**(*a*)Return True if *a* is unit normal.**lcm**(*a*, *b*)Return LCM of *a* and *b*.**quo**(*a*, *b*)Quotient of *a* and *b*, implies `__floordiv__`.**rem**(*a*, *b*)Remainder of *a* and *b*, implies `__mod__`.**property ring**

Return a ring associated with self.

class diofant.domains.simplesdomain.**SimpleDomain**

Base class for simple domains, e.g. ZZ, QQ.

inject(**gens*)

Inject generators into this domain.

class diofant.domains.compositedomain.**CompositeDomain**

Base class for composite domains, e.g. $\mathbb{Z}\mathbb{Z}[x]$, $\mathbb{Z}\mathbb{Z}(X)$.

inject(*symbols, front=False)

Inject generators into this domain.

class diofant.domains.characteristiczero.**CharacteristicZero**

Domain that has infinite number of elements.

4.9.2 Concrete Domains

class diofant.domains.**IntegerModRing**(order, dom)

General class for quotient rings over integers.

class diofant.domains.**FiniteField**(order, dom, modulus=None)

General class for finite fields.

class diofant.domains.**IntegerRing**

General class for integer rings.

property field

Return a field associated with self.

abstract finite_field(p)

Return a finite field.

abstract finite_ring(n)

Return a finite ring.

class diofant.domains.**RationalField**

General class for rational fields.

algebraic_field(*extension)

Return an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

class diofant.domains.**AlgebraicField**(dom, *ext)

A class for representing algebraic number fields.

algebraic_field(*extension)

Return an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

class diofant.domains.**RealAlgebraicField**(dom, *ext)

A class for representing real algebraic number fields.

class diofant.domains.**ComplexAlgebraicField**(dom, *ext)

A class for representing complex algebraic number fields.

class diofant.polys.rings.**PolynomialRing**(domain, symbols,
order=<diofant.polys.orderings.LexOrder
object>)

Return a multivariate polynomial ring.

drop(*gens)

Remove specified generators from this ring.

eject(*gens)

Remove specified generators from the ring and inject them into its domain.

property field

Returns a field associated with self.

gcdex(a, b)

Extended GCD of a and b.

half_gcdex(a, b)

Half extended GCD of a and b.

index(gen)

Compute index of gen in self.gens.

class diofant.polys.univar.**UnivarPolynomialRing**(domain, symbols, order=<diofant.polys.orderings.LexOrder object>)

A class for representing univariate polynomial rings.

dispersionset(p, q=None)

Compute the *dispersion set* of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

Examples

Note that the definition of the dispersion is not symmetric:

```
>>> R, x = ring('x', QQ)
```

```
>>> fp = x**4 - 3*x**2 + 1
>>> gp = fp.shift(-3)
```

```
>>> R.dispersionset(fp, gp)
{2, 3, 4}
>>> R.dispersionset(gp, fp)
set()
```

Computing the dispersion also works over field extensions:

```
>>> R, x = ring('x', QQ.algebraic_field(sqrt(5)))
```

```
>>> fp = x**2 + sqrt(5)*x - 1
>>> gp = x**2 + (2 + sqrt(5))*x + sqrt(5)
```

```
>>> R.dispersionset(fp, gp)
{2}
>>> R.dispersionset(gp, fp)
{1, 4}
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> D, a = ring('a', QQ)
>>> R, x = ring('x', D)
```

```
>>> fp = 4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x
>>> R.dispersionset(fp)
{0, 1}
```

References

- [MW94]
- [Koe98]
- [Abr71]
- [Man93]

class diofant.polys.fields.**FractionField**(*domain, symbols,*
order=<diofant.polys.orderings.LexOrder
object>)

A class for representing multivariate rational function fields.

class diofant.domains.**RealField**(*prec=53, dps=None, tol=None)*

Real numbers up to the given precision.

almosteq(*a, b, tolerance=None*)

Check if a and b are almost equal.

to_rational(*element, limit=True*)

Convert a real number to rational number.

class diofant.domains.**ComplexField**(*prec=53, dps=None, tol=None)*

Complex numbers up to the given precision.

almosteq(*a, b, tolerance=None*)

Check if a and b are almost equal.

class diofant.domains.**ExpressionDomain**

A class for arbitrary expressions.

class **Expression**(*ex*)

A class for elements of *ExpressionDomain* (page 431).

dtype

alias of *Expression* (page 431)

4.9.3 Implementation Domains

class diofant.domains.finitefield.**PythonFiniteField**(*order, modulus=None*)

Finite field based on Python's integers.

class diofant.domains.finitefield.**GMPYFiniteField**(*order, modulus=None*)

Finite field based on GMPY's integers.

class diofant.domains.integerring.**PythonIntegerRing**

Integer ring based on Python’s integers.

class diofant.domains.integerring.**GMPYIntegerRing**

Integer ring based on GMPY’s integers.

class diofant.domains.rationalfield.**PythonRationalField**

Rational field based on Python’s rationals.

class diofant.domains.rationalfield.**GMPYRationalField**

Rational field based on GMPY’s rationals.

4.9.4 Domain Elements

class diofant.domains.finitefield.**ModularInteger**(*rep*)

A class representing a modular integer.

property **is_primitive**

Test if this is a primitive element.

class diofant.domains.finitefield.**GaloisFieldElement**(*rep*)

A class representing a Galois field element.

4.10 Matrices

A module that handles matrices.

Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

class diofant.matrices.**Matrix**

alias of *MutableMatrix* (page 479)

4.10.1 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:

```
>>> init_printing(pretty_print=True, wrap_line=False, no_global=True)
>>> M = Matrix([[1, 0, 0], [0, 0, 0]])
>>> M

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

>>> Matrix([M, (0, 0, -1)])

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

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

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

```

(continues on next page)

(continued from previous page)

```
>>> Matrix([1, 2, 3])
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, Diofant 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])
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

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)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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)
```

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

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)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
>>> zeros(2)
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

```
>>> zeros(2, 5)
```

(continues on next page)

(continued from previous page)

```

[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]]
```

(continues on next page)

(continued from previous page)

```

[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:

```

>>> del M[0, :]
>>> M
[4 5 6]
[7 8 9]
>>> del M[:, 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

```

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]

```

(continues on next page)

(continued from previous page)

```

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:

```

>>> def f(x):
>>>     return 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!:

```

>>> 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

```

(continues on next page)

(continued from previous page)

```
>>> 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 Diofant, 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
[ sqrt(6)  -sqrt(3)  -sqrt(2) ]
[ 6        3        2 ]
```

(continues on next page)

(continued from previous page)

$$\begin{bmatrix} \frac{\sqrt{6}}{6} & \frac{-\sqrt{3}}{3} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{6}}{3} & \frac{\sqrt{3}}{3} & 0 \\ 0 & 0 & \sqrt{2} \end{bmatrix}$$

>>> R

$$\begin{bmatrix} \sqrt{6} & \frac{4 \cdot \sqrt{6}}{3} & 2 \cdot \sqrt{6} \\ 0 & \frac{\sqrt{3}}{3} & 0 \\ 0 & 0 & \sqrt{2} \end{bmatrix}$$

>>> 0*R

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

In addition to the solvers in the `solver.py` file, we can solve the system $Ax=b$ by passing the `b` vector to the matrix `A`'s `LUsolve` function. Here we'll cheat a little choose `A` and `x` then multiply to get `b`. Then we can solve for `x` and check that it's correct:

```
>>> A = Matrix([[2, 3, 5], [3, 6, 2], [8, 3, 6]])
>>> x = Matrix(3, 1, [3, 7, 5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[7]
[5]
```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to `False`. Let's take some vectors and orthogonalize them - one normalized and one not:

```
>>> L = [Matrix([2, 3, 5]), Matrix([3, 6, 2]), Matrix([8, 3, 6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)
```

Let's take a look at the vectors:

```
>>> for i in out1:
...     print(i)
Matrix([
[2],
[3],
```

(continues on next page)

(continued from previous page)

```
[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 diofant.matrices.matrices.**MatrixBase**

Base class for matrices.

property **C**

By-element conjugation.

property **D**

Return Dirac conjugate (if `self.rows == 4`).

Examples

```
>>> 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.
```

See also:

[conjugate \(page 449\)](#)

By-element conjugation

[H \(page 441\)](#)

Hermite conjugation

property H

Return Hermite conjugate.

Examples

```
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([
[0],
[1 + I],
[2],
[3]])
>>> m.H
Matrix([[0, 1 - I, 2, 3]])
```

See also:

[conjugate \(page 449\)](#)

By-element conjugation

[D \(page 440\)](#)

Dirac conjugation

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.

Examples

```

>>> 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

```

See also:

[cholesky](#) (page 447), [LUdecomposition](#) (page 442), [QRdecomposition](#) (page 443)

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.

Examples

```

>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True

```

See also:

[LDLdecomposition](#) (page 441), [lower_triangular_solve](#) (page 463), [upper_triangular_solve](#) (page 471), [cholesky_solve](#) (page 447), [diagonal_solve](#) (page 450), [LUsolve](#) (page 443), [QRsolve](#) (page 444), [pinv_solve](#) (page 466)

LUdecomposition(*iszerofunc*=<function _iszero>)

Returns the decomposition LU and the row swaps p.

Examples

```

>>> a = Matrix([[4, 3], [6, 3]])
>>> L, U, _ = a.LUdecomposition()
>>> L
Matrix([
[ 1, 0],
[ 3/2, 1]])
>>> U
Matrix([
[4, 3],
[0, -3/2]])

```

See also:

[cholesky](#) (page 447), [LDLdecomposition](#) (page 441), [QRdecomposition](#) (page 443), [LUdecomposition_Simple](#) (page 443), [LUdecompositionFF](#) (page 442), [LUsolve](#) (page 443)

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 442), [LUdecomposition_Simple](#) (page 443), [LUsolve](#) (page 443)

LUdecomposition_Simple(iszerofunc=<function _iszero>)

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 442), [LUdecompositionFF](#) (page 442), [LUsolve](#) (page 443)

LUsolve(rhs, iszerofunc=<function _iszero>)

Solve the linear system $Ax = rhs$ for x where $A = self$.

This is for symbolic matrices, for real or complex ones use `mpmath.lu_solve` or `mpmath.qr_solve`.

See also:

[lower_triangular_solve](#) (page 463), [upper_triangular_solve](#) (page 471), [cholesky_solve](#) (page 447), [diagonal_solve](#) (page 450), [LDLsolve](#) (page 442), [QRsolve](#) (page 444), [pinv_solve](#) (page 466), [LUdecomposition](#) (page 442)

QRdecomposition()

Return Q, R where $A = Q \cdot R$, Q is orthogonal and R is upper triangular.

Examples

This is the example from wikipedia:

```
>>> 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]])
```

See also:

[cholesky](#) (page 447), [LDLdecomposition](#) (page 441), [LUdecomposition](#) (page 442), [QRsolve](#) (page 444)

QRsolve(*b*)

Solve the linear system ' $Ax = b$ '.

'self' is the matrix 'A', the method argument is the vector 'b'. The method returns the solution vector 'x'. If 'b' is a matrix, the system is solved for each column of 'b' and the return value is a matrix of the same shape as 'b'.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact arithmetic, so you don't need to use QRsolve.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use `mpmath.qr_solve`.

See also:

[lower_triangular_solve](#) (page 463), [upper_triangular_solve](#) (page 471), [cholesky_solve](#) (page 447), [diagonal_solve](#) (page 450), [LDLsolve](#) (page 442), [LUsolve](#) (page 443), [pinv_solve](#) (page 466), [QRdecomposition](#) (page 443)

property 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.

<https://en.wikipedia.org/wiki/Adjugate>

See also:

[cofactorMatrix](#) (page 448), [transpose](#) (page 471), [berkowitz](#) (page 445)

atoms(types*)**

Returns the atoms that form the current object.

Examples

```
>>> Matrix([[x]])
Matrix([[x]])
>>> _atoms()
{x}
```

berkowitz()

The Berkowitz algorithm.

Given $N \times N$ 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 $N \times N$ and I is $N \times N$ identity matrix, then the following definition of characteristic polynomial is begin used:

$$\text{charpoly}(M) = \det(tI - M)$$

As a consequence, all polynomials generated by Berkowitz algorithm are monic.

```
>>> 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, Saarbrücken, 2006

See also:

[berkowitz_det](#) (page 446), [berkowitz_minors](#) (page 446), [berkowitz_charpoly](#) (page 445), [berkowitz_eigenvals](#) (page 446)

berkowitz_charpoly($x=\text{Dummy}('lambda')$, $\text{simplify}=\langle \text{function simplify} \rangle$)

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:

Examples

```
>>> 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
```

Be sure your provided `x` doesn't clash with existing symbols:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
-2*x + lambda**2 - lambda
>>> A.charpoly(x).as_expr()
Traceback (most recent call last):
GeneratorsError: polynomial ring and it's ground domain share generators
>>> A.charpoly(y).as_expr()
-2*x + y**2 - y
```

See also:

[berkowitz](#) (page 445)

berkowitz_det()

Computes determinant using Berkowitz method.

See also:

[det](#) (page 449), [berkowitz](#) (page 445)

berkowitz_eigenvals(flags)**

Computes eigenvalues of a Matrix using Berkowitz method.

See also:

[berkowitz](#) (page 445)

berkowitz_minors()

Computes principal minors using Berkowitz method.

See also:

[berkowitz](#) (page 445)

charpoly(x=Dummy('lambda'), simplify=<function simplify>)

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:

Examples

```
>>> 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
```

Be sure your provided x doesn't clash with existing symbols:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
-2*x + lambda**2 - lambda
>>> A.ccharpoly(x).as_expr()
Traceback (most recent call last):
GeneratorsError: polynomial ring and it's ground domain share generators
>>> A.charpoly(y).as_expr()
-2*x + y**2 - y
```

See also:

[berkowitz](#) (page 445)

`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

```
>>> 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]])
```

See also:

[LDLdecomposition](#) (page 441), [LUdecomposition](#) (page 442), [QRdecomposition](#) (page 443)

`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 463), [upper_triangular_solve](#) (page 471), [diagonal_solve](#) (page 450), [LDLsolve](#) (page 442), [LUsolve](#) (page 443), [QRsolve](#) (page 444), [pinv_solve](#) (page 466)

cofactor(*i, j, method='berkowitz'*)

Calculate the cofactor of an element.

See also:

[cofactorMatrix](#) (page 448), [minorEntry](#) (page 463), [minorMatrix](#) (page 463)

cofactorMatrix(*method='berkowitz'*)

Return a matrix containing the cofactor of each element.

See also:

[cofactor](#) (page 447), [minorEntry](#) (page 463), [minorMatrix](#) (page 463), [adjugate](#) (page 444)

col_insert(*pos, mti*)

Insert one or more columns at the given column position.

Examples

```
>>> 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]])
```

See also:

[row_insert](#) (page 468)

col_join(*bott*)

Concatenates two matrices along self's last and bott's first row

Examples

```
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[1, 1, 1]])
```

See also:

[row_join](#) (page 468)

condition_number()

Returns the condition number of a matrix.

This is the maximum singular value divided by the minimum singular value

Examples

```
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, Rational(1, 10)]])
>>> A.condition_number()
100
```

See also:

[singular_values](#) (page 469)

`conjugate()`

By-element conjugation.

`copy()`

Returns the copy of a matrix.

`cross(b)`

Return the cross product of *self* and *b* relaxing the condition of compatible dimensions: if each has 3 elements, a matrix of the same type and shape as *self* will be returned. If *b* has the same shape as *self* then common identities for the cross product (like $axb = -bxa$) will hold.

See also:

[dot](#) (page 451), [multiply](#) (page 463), [multiply_elementwise](#) (page 463)

`det(method='bareiss')`

Computes the matrix determinant using the method “method”.

Possible values for “method”:

bareiss ... det_bareis berkowitz ... berkowitz_det det_LU ...
det_LU_decomposition

See also:

[det_bareiss](#) (page 449), [berkowitz_det](#) (page 446), [det_LU_decomposition](#) (page 449)

`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), Hong R. Lee, B.David Saunders, Fraction Free Gaussian Elimination for Sparse Matrices, In Journal of Symbolic Computation, Volume 19, Issue 5, 1995, Pages 393-402, ISSN 0747-7171, <https://www.sciencedirect.com/science/article/pii/S074771718571022X>.

See also:

[det](#) (page 449), [det_bareiss](#) (page 449), [berkowitz_det](#) (page 446)

`det_bareiss()`

Compute matrix determinant using Bareiss’ 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), Hong R. Lee, B.David Saunders, Fraction Free Gaussian Elimination for Sparse Matrices, In Journal of Symbolic Computation, Volume 19, Issue 5, 1995, Pages 393-402, ISSN 0747-7171, <https://www.sciencedirect.com/science/article/pii/S074771718571022X>.

See also:

det (page 449), *berkowitz_det* (page 446)

diagonal_solve(rhs)

Solves $Ax = B$ efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

Examples

```
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True
```

See also:

lower_triangular_solve (page 463), *upper_triangular_solve* (page 471), *cholesky_solve* (page 447), *LDLsolve* (page 442), *LUsolve* (page 443), *QRsolve* (page 444), *pinv_solve* (page 466)

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.

Examples

```
>>> 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]])
```

See also:

is_diagonal (page 456), *is_diagonalizable* (page 456)

diff(*args)

Calculate the derivative of each element in the matrix.

Examples

```
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0]
[0, 0]])
```

See also:

[integrate](#) (page 453), [limit](#) (page 463)

`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).

Examples

```
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = [1, 1, 1]
>>> M[0, :].dot(v)
6
>>> M[:, 0].dot(v)
12
>>> M.dot(v)
[6, 15, 24]
```

See also:

[cross](#) (page 449), [multiply](#) (page 463), [multiply_elementwise](#) (page 463)

`dual()`

Returns the dual of a matrix, which is:

$(1/2) * \text{levicivita}(i, j, k, l) * M(k, l)$ summed over indices k and l

Since the levicivita method is anti_symmetric for any pairwise exchange of indices, the dual of a symmetric matrix is the zero matrix. Strictly speaking the dual defined here assumes that the 'matrix' M is a contravariant anti_symmetric second rank tensor, so that the dual is a covariant second rank tensor.

`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(*dps=15, **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

```
>>> 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 $-n \leq i < n$ where n is the number of rows or columns.

Examples

```
>>> 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]])
```

property free_symbols

Returns the free symbols within the matrix.

Examples

```
>>> Matrix([[x], [1]]).free_symbols
{x}
```

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

```
>>> 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

```
>>> 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

```
>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2]])
```

integrate(*args)

Integrate each element of the matrix.

Examples

```
>>> 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]])
```

See also:

[limit](#) (page 463), [diff](#) (page 450)

inv(*method=None*, ***kwargs*)

Returns the inverse of the matrix.

Parameters

method ({'GE', 'LU', 'ADJ', 'CH', 'LDL'} or *None*) – Selects algorithm for inversion. For dense matrices available {'GE', 'LU', 'ADJ'}, default is 'GE'. For sparse: {'CH', 'LDL'}, default is 'LDL'.

Raises

ValueError – If the determinant of the matrix is zero.

See also:

[inverse_LU](#) (page 455), [inverse_GE](#) (page 454), [inverse_ADJ](#) (page 454)

inv_mod(*m*)

Returns the inverse of the matrix K (mod m), if it exists.

Method to find the matrix inverse of K (mod m) implemented in this function:

- Compute $\text{adj}(K) = \text{cof}(K)^t$, the adjoint matrix of K .
- Compute $r = 1/\det(K)$ (mod m).
- $K^{-1} = r \cdot \text{adj}(K)$ (mod m).

Examples

```
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([
[3, 1],
[4, 2]])
>>> A.inv_mod(3)
Matrix([
[1, 1],
[0, 1]])
```

inverse_ADJ(*iszerofunc=<function _iszero>*)

Calculates the inverse using the adjugate matrix and a determinant.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 454), [inverse_LU](#) (page 455), [inverse_GE](#) (page 454)

inverse_GE(iszerofunc=<function _iszero>)

Calculates the inverse using Gaussian elimination.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 454), [inverse_LU](#) (page 455), [inverse_ADJ](#) (page 454)

inverse_LU(iszerofunc=<function _iszero>)

Calculates the inverse using LU decomposition.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 454), [inverse_GE](#) (page 454), [inverse_ADJ](#) (page 454)

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 Diofant 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

```
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1]
[-1, 0]])
>>> m.is_anti_symmetric()
True
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
>>> m
Matrix([
[ 0, 0, x]
[-y, 0, 0]])
>>> m.is_anti_symmetric()
False
```

```
>>> 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.

Examples

```
>>> 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
```

See also:

[`is_lower`](#) (page 457), [`is_upper`](#) (page 460), [`is_diagonalizable`](#) (page 456), [`diagonalize`](#) (page 450)

`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.

Examples

```
>>> 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
```

(continues on next page)

(continued from previous page)

```

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

```

See also:

[*is_diagonal*](#) (page 456), [*diagonalize*](#) (page 450)

property is_hermitian

Checks if the matrix is Hermitian.

In a Hermitian matrix element i,j is the complex conjugate of element j,i .

Examples

```

>>> a = Matrix([[1, I], [-I, 1]])
>>> a
Matrix([
[1, I],
[-I, 1]])
>>> a.is_hermitian
True
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
>>> a[0, 0] = x
>>> a.is_hermitian
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False

```

property is_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.

Examples

```

>>> 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

```

```
>>> 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
```

See also:

[is_upper](#) (page 460), [is_diagonal](#) (page 456), [is_lower_hessenberg](#) (page 458)

property `is_lower_hessenberg`

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

Examples

```
>>> 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
```

See also:

[is_upper_hessenberg](#) (page 460), [is_lower](#) (page 457)

`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

```
>>> 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
```

property `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

```
>>> 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

```
>>> 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

```
>>> 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
```

```
>>> 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
```

property `is_upper`

Check if matrix is an upper triangular matrix. True can be returned even if the matrix is not square.

Examples

```
>>> 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
```

See also:

[`is_lower`](#) (page 457), [`is_diagonal`](#) (page 456), [`is_upper_hessenberg`](#) (page 460)

property `is_upper_hessenberg`

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

Examples

```
>>> 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
```

See also:

[`is_lower_hessenberg`](#) (page 458), [`is_upper`](#) (page 460)

property `is_zero`

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be `None`

Examples

```
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([[1]])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero
```

`jacobian(X)`

Calculates the Jacobian matrix (derivative of a vectorial function).

Parameters

- **self** (vector of expressions representing functions $f_i(x_1, \dots, x_n)$.)
- **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).**

Examples

```
>>> from diofant.abc import phi, rho
>>> 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)]])
```

See also:

[`diofant.matrices.dense.hessian`](#) (page 475), [`diofant.matrices.dense.wronskian`](#) (page 476)

`jordan_cells(calc_transformation=True)`

Return a list of Jordan cells of current matrix. This list shape Jordan matrix J .

If `calc_transformation=False`, then transformation P such that

$$J = P^{-1} \cdot M \cdot P$$

will not be calculated.

Examples

```
>>> m = Matrix([ [+6, 5, -2, -3],
...               [-3, -1, 3, 3],
...               [+2, 1, -2, -3],
...               [-1, 1, 5, 5] ])
```

```
>>> Jcells, P = m.jordan_cells()
>>> Jcells[0]
Matrix([
[2, 1],
[0, 2]])
>>> Jcells[1]
Matrix([
[2, 1],
[0, 2]])
```

See also:

[*jordan_form*](#) (page 462)

jordan_form(*calc_transformation=True*)

Return Jordan form J of current matrix.

Also (if *calc_transformation=True*) the transformation P such that

$$J = P^{-1} \cdot M \cdot P$$

and the jordan blocks forming J will be calculated.

Examples

```
>>> m = Matrix([ [+6, 5, -2, -3],
...               [-3, -1, 3, 3],
...               [+2, 1, -2, -3],
...               [-1, 1, 5, 5] ])
>>> J, P = m.jordan_form()
>>> J
Matrix([
[2, 1, 0, 0],
[0, 2, 0, 0],
[0, 0, 2, 1],
[0, 0, 0, 2]])
```

See also:

[*jordan_cells*](#) (page 461)

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 462)

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 462)**limit(*args)**

Calculate the limit of each element in the matrix.

Examples

```
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

See also:[integrate](#) (page 453), [diff](#) (page 450)**lower_triangular_solve(rhs)**Solves $Ax = B$, where A is a lower triangular matrix.**See also:**[upper_triangular_solve](#) (page 471), [cholesky_solve](#) (page 447), [diagonal_solve](#) (page 450), [LDLsolve](#) (page 442), [LUsolve](#) (page 443), [QRsolve](#) (page 444), [pinv_solve](#) (page 466)**minorEntry(i, j, method='berkowitz')**

Calculate the minor of an element.

See also:[minorMatrix](#) (page 463), [cofactor](#) (page 447), [cofactorMatrix](#) (page 448)**minorMatrix(i, j)**

Creates the minor matrix of a given element.

See also:[minorEntry](#) (page 463), [cofactor](#) (page 447), [cofactorMatrix](#) (page 448)**multiply(b)**Returns $\text{self} * b$ **See also:**[dot](#) (page 451), [cross](#) (page 449), [multiply_elementwise](#) (page 463)**multiply_elementwise(b)**

Return the Hadamard product (elementwise product) of A and B

Examples

```
>>> 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])
```

See also:

[cross](#) (page 449), [dot](#) (page 451), [multiply](#) (page 463)

`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)

Examples

```
>>> x = Symbol('x', real=True)
>>> v = Matrix([cos(x), sin(x)])
>>> trigsimp(v.norm())
1
>>> v.norm(10)
root(sin(x)**10 + cos(x)**10, 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
```

See also:

[normalized](#) (page 464)

`normalized()`

Return the normalized version of `self`.

See also:[norm](#) (page 464)**nullspace**(*simplify=False, iszerofunc=<function _iszero>*)

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.

Examples

```
>>> M = eye(3)
>>> M.permuteBkwd([[0, 1], [0, 2]])
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

See also:[permuteFwd](#) (page 465)**permuteFwd**(*perm*)

Permute the rows of the matrix with the given permutation.

Examples

```
>>> M = eye(3)
>>> M.permuteFwd([[0, 1], [0, 2]])
Matrix([
[0, 0, 1],
[1, 0, 0],
[0, 1, 0]])
```

See also:[permuteBkwd](#) (page 465)**pinv**()

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

Examples

```
>>> Matrix([[1, 2, 3], [4, 5, 6]]).pinv()
Matrix([
[-17/18, 4/9],
[-1/9, 1/9],
[13/18, -2/9]])
```

See also:[diofant.matrices.matrices.MatrixBase.inv](#) (page 454), [pinv_solve](#) (page 466)

References

- https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse

pinv_solve(*B*, *arbitrary_matrix*=None)

Solve $Ax = B$ using the Moore-Penrose pseudoinverse.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of *arbitrary_matrix*. If no solutions exist, the least-squares solution is returned.

Parameters

- **B** (*Matrix*) - The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.
- **arbitrary_matrix** (*Matrix*) - If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of *wn_m* will be used, with n and m being row and column position of each symbol.

Returns

x (*Matrix*) - The matrix that will satisfy $Ax = B$. Will have as many rows as matrix A has columns, and as many columns as matrix B.

Examples

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([
[ -w0 0/6 - w1 0/3 + w2 0/6 - 55/18],
[ -w0 0/3 + 2*w1 0/3 - w2 0/3 + 1/9],
[ -w0 0/6 - w1 0/3 + w2 0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([
[ -55/18],
[ 1/9],
[ 59/18]])
```

See also:

lower_triangular_solve (page 463), *upper_triangular_solve* (page 471), *cholesky_solve* (page 447), *diagonal_solve* (page 450), *LDLsolve* (page 442), *LU_solve* (page 443), *QRsolve* (page 444), *pinv* (page 465)

Notes

This may return either exact solutions or least squares solutions. To determine which, check `A * A.pinv() * B == B`. It will be True if exact solutions exist, and False if only a least-squares solution exists. Be aware that the left hand side of that equation may need to be simplified to correctly compare to the right hand side.

References

- https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse#Obtaining_all_solutions_of_a_linear_system

print_nonzero(symb='X')

Shows location of non-zero entries for fast shape lookup.

Examples

```
>>> 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

```
>>> V = Matrix([sqrt(3)/2, Rational(1, 2)])
>>> x = Matrix([[1, 0]])
>>> V.project(x)
Matrix([[sqrt(3)/2, 0]])
>>> V.project(-x)
Matrix([[sqrt(3)/2, 0]])
```

rank(iszerofunc=<function _iszero>, simplify=False)

Returns the rank of a matrix

```
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2
```

replace(F, G, exact=False)

Replaces Function F in Matrix entries with Function G.

Examples

```
>>> 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.

Examples

```
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([
[0, 0, 0],
[1, 1, 1],
[0, 0, 0]])
```

See also:

[*col_insert*](#) (page 448)

row_join(*rhs*)

Concatenates two matrices along self's last and rhs's first column

Examples

```
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.row_join(V)
Matrix([
[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

See also:

[*col_join*](#) (page 448)

rref(*iszerofunc*=<function *_iszero*>, *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 Diofant simplify function) or pass a custom simplify function.

Examples

```
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([
[1, 0],
[0, 1]]), [0, 1])
```

property shape

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

Examples

```
>>> M = zeros(2, 3)
>>> M.shape
(2, 3)
>>> M.rows
2
>>> M.cols
3
```

simplify(ratio=1.7, measure=<function count_ops>)

Apply simplify to each element of the matrix.

Examples

```
>>> 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

Examples

```
>>> 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]
```

See also:

[condition_number](#) (page 448)

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.

Examples

```
>>> 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().evalf(2)
0.58
```

If a different xy is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().evalf(2)
1.5
```

See also:

[inv](#) (page 454)

subs(*args, **kwargs)

Return a new matrix with subs applied to each entry.

Examples

```
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> .subs({x: y})
Matrix([[y]])
>>> Matrix(_).subs({y: x})
Matrix([[x]])
```

table(printer, rowstart='[', rowend=']', rowsep='\n', colsep=', ', align='right')

String form of Matrix as a table.

printer is the printer to use for on the elements (generally something like Str-Printer())

rowstart is the string used to start each row (by default '[').

rowend is the string used to end each row (by default ']').

rowsep is the string used to separate rows (by default a newline).

colsep is the string used to separate columns (by default ', ').

align defines how the elements are aligned. Must be one of 'left', 'right', or 'center'. You can also use '<', '>', and '^' to mean the same thing, respectively.

This is used by the string printer for Matrix.

Examples

```
>>> from diofant.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(M.table(printer, colsep=' '))
[ 1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1, 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{ 1, 2}
{-33, 4}
```

trace()

Returns the trace of a matrix.

transpose()

Matrix transposition.

upper_triangular_solve(rhs)

Solves $Ax = B$, where A is an upper triangular matrix.

See also:

[lower_triangular_solve](#) (page 463), [cholesky_solve](#) (page 447), [diagonal_solve](#) (page 450), [LDLsolve](#) (page 442), [LUsolve](#) (page 443), [QRsolve](#) (page 444), [pinv_solve](#) (page 466)

values()

Return non-zero values of self.

vec()

Return the Matrix converted into a one column matrix by stacking columns

Examples

```
>>> m = Matrix([[1, 3], [2, 4]])
>>> m
Matrix([
[1, 3],
[2, 4]])
>>> m.vec()
Matrix([
[1],
[2],
[3],
[4]])
```

See also:

[vech](#) (page 472)

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

Examples

```
>>> m = Matrix([[1, 2], [2, 3]])
>>> m
Matrix([
[1, 2],
[2, 3]])
>>> m.vech()
Matrix([
[1],
[2],
[3]])
>>> m.vech(diagonal=False)
Matrix([[2]])
```

See also:

[vec](#) (page 471)

classmethod vstack(*args)

Return a matrix formed by joining args vertically (i.e. by repeated application of `col_join`).

Examples

```
>>> Matrix.vstack(eye(2), 2*eye(2))
Matrix([
[1, 0],
[0, 1],
[2, 0],
[0, 2]])
```

xreplace(rule)

Return a new matrix with `xreplace` applied to each entry.

Examples

```
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _xreplace({x: y})
Matrix([[y]])
>>> Matrix(_).xreplace({y: x})
Matrix([[x]])
```

Matrix Exceptions Reference

class diofant.matrices.matrices.**MatrixError**

Generic matrix error.

class diofant.matrices.matrices.**ShapeError**

Wrong matrix shape.

class diofant.matrices.matrices.**NonSquareMatrixError**

Raised when a square matrix is expected.

Matrix Functions Reference

diofant.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

```
>>> M = Matrix([[1, 2], [3, 4]]) # a Mutable Matrix
>>> IM = ImmutableMatrix([[1, 2], [3, 4]])
>>> classof(M, IM)
<class 'diofant.matrices.immutable.ImmutableMatrix'>
```

diofant.matrices.dense.**matrix_multiply_elementwise**(*A*, *B*)

Return the Hadamard product (elementwise product) of *A* and *B*

```
>>> 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:

[*diofant.matrices.dense.DenseMatrix.__mul__*](#) (page 480)

diofant.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:

[*diofant.matrices.dense.ones*](#) (page 473), [*diofant.matrices.dense.eye*](#) (page 474), [*diofant.matrices.dense.diag*](#) (page 474)

`diofant.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:

[`diofant.matrices.dense.zeros`](#) (page 473), [`diofant.matrices.dense.eye`](#) (page 474), [`diofant.matrices.dense.diag`](#) (page 474)

`diofant.matrices.dense.eye(n, cls=None)`

Create square identity matrix *n* x *n*

See also:

[`diofant.matrices.dense.diag`](#) (page 474), [`diofant.matrices.dense.zeros`](#) (page 473), [`diofant.matrices.dense.ones`](#) (page 473)

`diofant.matrices.dense.diag(*values, **kwargs)`

Create a sparse, diagonal matrix from a list of diagonal values.

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

```
>>> 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:

```
>>> 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],
```

(continues on next page)

(continued from previous page)

```
[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 'diofant.matrices.dense.MutableDenseMatrix'>
>>> type(diag(1, cls=ImmutableMatrix))
<class 'diofant.matrices.immutable.ImmutableMatrix'>
```

See also:

[diofant.matrices.dense.eye](#) (page 474)

`diofant.matrices.dense.jordan_cell(eigenval, n)`

Create matrix of Jordan cell kind:

Examples

```
>>> jordan_cell(x, 4)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

`diofant.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.

Examples

```
>>> f = Function('f')(x, y)
>>> g1 = Function('g')(x, y)
>>> g2 = x**2 + 3*y
>>> pprint(hessian(f, (x, y), [g1, g2]))
[
  0      0       $\frac{\partial}{\partial x}(g(x, y))$    $\frac{\partial}{\partial y}(g(x, y))$ 
  0      0       $2 \cdot x$       3
]
```

(continues on next page)

(continued from previous page)

$$\begin{bmatrix} \frac{\partial}{\partial x}(g(x, y)) & 2 \cdot x & \frac{\partial^2}{\partial x^2}(f(x, y)) & \frac{\partial^2}{\partial y \partial x}(f(x, y)) \\ \frac{\partial}{\partial y}(g(x, y)) & 3 & \frac{\partial^2}{\partial y \partial x}(f(x, y)) & \frac{\partial^2}{\partial y^2}(f(x, y)) \end{bmatrix}$$

References

https://en.wikipedia.org/wiki/Hessian_matrix

See also:

diofant.matrices.matrices.MatrixBase.jacobian (page 461), *diofant.matrices.dense.wronskian* (page 476)

`diofant.matrices.dense.GramSchmidt(vlist, orthonormal=False)`

Apply the Gram-Schmidt process to a set of vectors.

see: https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process

`diofant.matrices.dense.wronskian(functions, var, method='bareiss')`

Compute Wronskian for [] of functions

$$W(f_1, \dots, f_n) = \begin{vmatrix} f_1 & f_2 & \dots & f_n \\ f_1' & f_2' & \dots & f_n' \\ \vdots & \vdots & \ddots & \vdots \\ f_1^{(n)} & f_2^{(n)} & \dots & f_n^{(n)} \end{vmatrix} = \begin{vmatrix} D^{(n)}(f_1) & D^{(n)}(f_2) & \dots & D^{(n)}(f_n) \end{vmatrix}$$

see: <https://en.wikipedia.org/wiki/Wronskian>

See also:

diofant.matrices.matrices.MatrixBase.jacobian (page 461), *diofant.matrices.dense.hessian* (page 475)

`diofant.matrices.dense.casoratian(seqs, n, zero=True)`

Given linear difference operator L of order ' k ' and homogeneous equation $Ly = 0$ we want to compute kernel of L , which is a set of ' k ' sequences: $a(n)$, $b(n)$, ... $z(n)$.

Solutions of L are linearly independent iff their Casoratian, denoted as $C(a, b, \dots, z)$, do not vanish for $n = 0$.

Casoratian is defined by $k \times k$ determinant:

$$\begin{vmatrix} + & a(n) & b(n) & \dots & z(n) & + \\ & a(n+1) & b(n+1) & \dots & z(n+1) & \\ & \vdots & \vdots & \ddots & \vdots & \\ + & a(n+k-1) & b(n+k-1) & \dots & z(n+k-1) & + \end{vmatrix}$$

It proves very useful in `rsolve_hyper()` where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```
>>> n = Symbol('n', integer=True)
```

Exponential and factorial are linearly independent:

```
>>> casoratian([2**n, factorial(n)], n) != 0
True
```

```
diofant.matrices.dense.randMatrix(r, c=None, min=0, max=99, seed=None,
                                   symmetric=False, percent=100)
```

Create random matrix with dimensions $r \times c$. If c is omitted the matrix will be square. If `symmetric` is `True` the matrix must be square. If `percent` is less than 100 then only approximately the given percentage of elements will be non-zero.

Examples

```
>>> randMatrix(3, seed=0)
Matrix([
[49, 97, 53],
[ 5, 33, 65],
[62, 51, 38]])
>>> randMatrix(3, 2, seed=0)
Matrix([
[49, 97],
[53,  5],
[33, 65]])
>>> randMatrix(3, 3, 0, 2, seed=0)
Matrix([
[1, 1, 0],
[1, 2, 1],
[1, 1, 1]])
>>> randMatrix(3, symmetric=True, seed=0)
Matrix([
[49, 97, 53],
[97,  5, 33],
[53, 33, 65]])
>>> 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, seed=0)
Matrix([
[ 0,  0,  5],
[33,  0,  0],
[65, 53, 33]])
```

Numpy Utility Functions Reference

`diofant.matrices.dense.list2numpy(l, dtype=<class 'object'>)`

Converts python list of Diofant expressions to a NumPy array.

See also:

[`diofant.matrices.dense.matrix2numpy`](#) (page 477)

`diofant.matrices.dense.matrix2numpy(m, dtype=<class 'object'>)`

Converts Diofant's matrix to a NumPy array.

See also:

[`diofant.matrices.dense.list2numpy`](#) (page 477)

`diofant.matrices.dense.rot_axis1(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 1-axis.

Examples

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]])
```

See also:

[**`diofant.matrices.dense.rot_axis2` \(page 478\)**](#)

Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis

[**`diofant.matrices.dense.rot_axis3` \(page 478\)**](#)

Returns a rotation matrix for a rotation of θ (in radians) about the 3-axis

`diofant.matrices.dense.rot_axis2(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis.

Examples

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]])
```

See also:

[**`diofant.matrices.dense.rot_axis1` \(page 478\)**](#)

Returns a rotation matrix for a rotation of θ (in radians) about the 1-axis

[**`diofant.matrices.dense.rot_axis3` \(page 478\)**](#)

Returns a rotation matrix for a rotation of θ (in radians) about the 3-axis

`diofant.matrices.dense.rot_axis3(theta)`

Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.

Examples

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]])
```

See also:

[`diofant.matrices.dense.rot_axis1`](#) (page 478)

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

[`diofant.matrices.dense.rot_axis2`](#) (page 478)

Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

`diofant.matrices.matrices.a2idx(j, n=None)`

Return integer after making positive and validating against n.

4.10.2 Dense Matrices

`diofant.matrices.dense.MutableMatrix`

alias of [`MutableDenseMatrix`](#) (page 482)

Matrix Class Reference

class `diofant.matrices.dense.DenseMatrix`

A dense matrix base class.

`__getitem__(key)`

Return portion of self defined by key. If the key involves a slice then a list will be returned (if key is a single slice) or a matrix (if key was a tuple involving a slice).

Examples

```
>>> m = Matrix([[1, 2 + I], [3, 4]])
```

If the key is a tuple that doesn't involve a slice then that element is returned:

```
>>> m[1, 0]
3
```

When a tuple key involves a slice, a matrix is returned. Here, the first column is selected (all rows, column 0):

```
>>> m[:, 0]
Matrix([
[1]
[3]])
```

If the slice is not a tuple then it selects from the underlying list of elements that are arranged in row order and a list is returned if a slice is involved:

```
>>> m[0]
1
>>> m[:, 2]
[1, 3]
```

__mul__(other)

Return self*other.

applyfunc(f)

Apply a function to each element of the matrix.

Examples

```
>>> m = Matrix(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

```
>>> 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.

Examples

```
>>> 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
```

See also:

[*diofant.core.expr.Expr.equals*](#) (page 67)

classmethod **eye**(*n*)

Return an $n \times n$ identity matrix.

reshape(*rows*, *cols*)

Reshape the matrix. Total number of elements must remain the same.

Examples

```
>>> m = Matrix(2, 3, lambda i, j: 1)
>>> m
Matrix([
[1, 1, 1],
[1, 1, 1]])
>>> m.reshape(1, 6)
Matrix([[1, 1, 1, 1, 1, 1]])
>>> m.reshape(3, 2)
Matrix([
[1, 1],
[1, 1],
[1, 1]])
```

tolist()

Return the Matrix as a nested Python list.

Examples

```
>>> m = Matrix(3, 3, range(9))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
>>> m.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> ones(3, 6).tolist()
[[1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> ones(0, 3).tolist()
[]
```

classmethod `zeros(r, c=None)`

Return an $r \times c$ matrix of zeros, square if c is omitted.

class `diofant.matrices.dense.MutableDenseMatrix(*args, **kwargs)`

A mutable version of the dense matrix.

col_op(j, f)

In-place operation on col j using two-arg functor whose args are interpreted as $(\text{self}[i, j], i)$.

Examples

```
>>> 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]])
```

See also:

[`diofant.matrices.dense.MutableDenseMatrix.row_op`](#) (page 483)

col_swap(i, j)

Swap the two given columns of the matrix in-place.

Examples

```
>>> M = Matrix([[1, 0], [1, 0]])
>>> M
Matrix([
[1, 0],
[1, 0]])
>>> M.col_swap(0, 1)
>>> M
Matrix([
[0, 1],
[0, 1]])
```

See also:

[`diofant.matrices.dense.MutableDenseMatrix.row_swap`](#) (page 484)

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.

Examples

```

>>> 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]])

```

See also:

[`diofant.matrices.dense.MutableDenseMatrix.copyin_matrix`](#) (page 483)

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.

Examples

```

>>> 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]])

```

See also:

[`diofant.matrices.dense.MutableDenseMatrix.copyin_list`](#) (page 482)

fill(value)

Fill the matrix with the scalar value.

See also:

[`diofant.matrices.dense.zeros`](#) (page 473), [`diofant.matrices.dense.ones`](#) (page 473)

row_op(i, f)

In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i, j], j).

Examples

```
>>> 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]])
```

See also:

[*diofant.matrices.dense.MutableDenseMatrix.zip_row_op*](#) (page 484), [*diofant.matrices.dense.MutableDenseMatrix.col_op*](#) (page 482)

row_swap(i, j)

Swap the two given rows of the matrix in-place.

Examples

```
>>> M = Matrix([[0, 1], [1, 0]])
>>> M
Matrix([
[0, 1],
[1, 0]])
>>> M.row_swap(0, 1)
>>> M
Matrix([
[1, 0],
[0, 1]])
```

See also:

[*diofant.matrices.dense.MutableDenseMatrix.col_swap*](#) (page 482)

simplify(ratio=1.7, measure=<function count_ops>)

Applies simplify to the elements of a matrix in place.

This is a shortcut for `M.applyfunc(lambda x: simplify(x, ratio, measure))`

See also:

[*diofant.simplify.simplify.simplify*](#) (page 580)

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])`.

Examples

```
>>> 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]])
```

See also:

[*diofant.matrices.dense.MutableDenseMatrix.row_op*](#) (page 483), [*diofant.matrices.dense.MutableDenseMatrix.col_op*](#) (page 482)

ImmutableMatrix Class Reference

class diofant.matrices.immutable.ImmutableMatrix(*args, **kwargs)

Create an immutable version of a matrix.

Examples

```
>>> 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
```

property C

By-element conjugation.

adjoint()

Conjugate transpose or Hermitian conjugation.

as_mutable()

Returns a mutable version of this matrix

Examples

```
>>> 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]])
```

conjugate()

By-element conjugation.

diff(*args)

Calculate the derivative of each element in the matrix.

Examples

```
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

See also:

[integrate](#) (page 499), [limit](#) (page 499)

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.

Examples

```
>>> 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
```

See also:

[*diofant.core.expr.Expr.equals*](#) (page 67)

integrate(*args)

Integrate each element of the matrix.

Examples

```
>>> 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]])
```

See also:

[*limit*](#) (page 499), [*diff*](#) (page 498)

limit(*args)

Calculate the limit of each element in the matrix.

Examples

```
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

See also:

[*integrate*](#) (page 499), [*diff*](#) (page 498)

4.10.3 Sparse Matrices

class diofant.matrices.sparse.MutableSparseMatrix(*args)

A sparse matrix (a matrix with a large number of zero elements).

Examples

```
>>> SparseMatrix(2, 2, range(4))
Matrix([
[0, 1],
[2, 3]])
>>> SparseMatrix(2, 2, {(1, 1): 2})
Matrix([
[0, 0],
[0, 2]])
```

See also:

[*diofant.matrices.dense.DenseMatrix*](#) (page 479)

col_join(bott)

Returns B augmented beneath A (row-wise joining):

```
[A]
[B]
```

Examples

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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_join(*rhs*)

Returns B appended after A (column-wise augmenting):

```
[A B]
```

Examples

```

>>> 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*).

Examples

```

>>> 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]])

```

See also:

[`zip_row_op`](#) (page 489), [`col_op`](#) (page 487)

`row_swap(i, j)`

Swap, in place, columns *i* and *j*.

Examples

```

>>> 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]`).

Examples

```
>>> 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]])
```

See also:

[row_op](#) (page 489), [col_op](#) (page 487)

`diofant.matrices.sparse.SparseMatrix`

alias of [MutableSparseMatrix](#) (page 487)

class `diofant.matrices.sparse.SparseMatrixBase(*args)`

A sparse matrix base class.

property `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

```
>>> 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
```

property `RL`

Alternate faster representation

`add(b)`

Add two sparse matrices with dictionary representation.

Examples

```
>>> 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
{}
```

See also:

[multiply](#) (page 494)

applyfunc(f)

Apply a function to each element of the matrix.

Examples

```
>>> 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

```
>>> 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

```
>>> A = SparseMatrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T == A
True
```

col_list()

Returns a column-sorted list of non-zero elements of the matrix.

Examples

```
>>> SparseMatrix(((1, 2), (3, 4)))
Matrix([
[1, 2],
[3, 4]])
>>> CL
[(0, 0, 1), (1, 0, 3), (0, 1, 2), (1, 1, 4)]
```

See also:

[diofant.matrices.sparse.MutableSparseMatrix.col_op](#) (page 487), [row_list](#) (page 494)

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 $-n \leq i < n$ where n is the number of rows or columns.

Examples

```
>>> 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]])
```


classmethod `eye(n)`

Return an $n \times n$ identity matrix.

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> A = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
```

property `is_hermitian`

Checks if the matrix is Hermitian.

In a Hermitian matrix element i,j is the complex conjugate of element j,i .

Examples

```
>>> a = SparseMatrix([[1, I], [-I, 1]])
>>> a
Matrix([
[ 1, I]
[-I, 1]])
>>> a.is_hermitian
True
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
>>> a[0, 0] = x
>>> a.is_hermitian
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False
```

is_symmetric(*simplify=True*)

Return True if self is symmetric.

Examples

```
>>> 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.

Examples

```
>>> 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])
```

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Grondelle (1999)

multiply(b)

Fast multiplication exploiting the sparsity of the matrix.

Examples

```
>>> A, B = SparseMatrix(ones(4, 3)), SparseMatrix(ones(3, 4))
>>> A.multiply(B) == 3*ones(4)
True
```

See also:

[add](#) (page 490)

nnz()

Returns the number of non-zero elements in Matrix.

reshape(rows, cols)

Reshape matrix while retaining original size.

Examples

```
>>> S = SparseMatrix(4, 2, range(8))
>>> S.reshape(2, 4)
Matrix([
[0, 1, 2, 3]
[4, 5, 6, 7]])
```

row_list()

Returns a row-sorted list of non-zero elements of the matrix.

Examples

```
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([
[1, 2]
[3, 4]])
>>> aRL
[(0, 0, 1), (0, 1, 2), (1, 0, 3), (1, 1, 4)]
```

See also:

[`diofant.matrices.sparse.MutableSparseMatrix.row_op`](#) (page 489), [`col_list`](#) (page 492)

row_structure_symbolic_cholesky()

Symbolic cholesky factorization, for pre-determination of the non-zero structure of the Cholesky factorization.

Examples

```
>>> 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]]
```

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Grondelle (1999)

scalar_multiply(*scalar*)

Scalar element-wise multiplication.

solve(*rhs*, *method*='LDL')

Return solution to $\text{self} * \text{soln} = \text{rhs}$ using given inversion method.

See also:

[`diofant.matrices.matrices.MatrixBase.inv`](#) (page 454)

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.

Examples

```
>>> 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().evalf(2)
0.58
```

If a different xy is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().evalf(2)
1.5
```

See also:

[*diofant.matrices.matrices.MatrixBase.inv*](#) (page 454)

tolist()

Convert this sparse matrix into a list of nested Python lists.

Examples

```
>>> 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 $r \times c$ matrix of zeros, square if c is omitted.

ImmutableSparseMatrix Class Reference

class `diofant.matrices.immutable.ImmutableSparseMatrix(*args, **kwargs)`

Create an immutable version of a sparse matrix.

Examples

```
>>> 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

```
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.subs({x: y})
Matrix([[y]])
>>> Matrix(_).subs({y: x})
Matrix([[x]])
```

xreplace(rule)

Return a new matrix with xreplace applied to each entry.

Examples

```
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.xreplace({x: y})
Matrix([[y]])
>>> Matrix(_).xreplace({y: x})
Matrix([[x]])
```

4.10.4 Immutable Matrices

The standard Matrix class in Diofant is mutable. This is important for performance reasons but means that standard matrices cannot interact well with the rest of Diofant. This is because the *Basic* (page 46) object, from which most Diofant classes inherit, is immutable.

The mission of the *ImmutableMatrix* (page 498) 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* (page 46) and can thus interact more naturally with the rest of Diofant. *ImmutableMatrix* (page 498) also inherits from *MatrixExpr* (page 500), allowing it to interact freely with Diofant's Matrix Expression module.

You can turn any Matrix-like object into an *ImmutableMatrix* (page 498) by calling the constructor

```
>>> 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 diofant.matrices.immutable.ImmutableMatrix(*args, **kwargs)

Create an immutable version of a matrix.

Examples

```
>>> 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
```

property C

By-element conjugation.

adjoint()

Conjugate transpose or Hermitian conjugation.

as_mutable()

Returns a mutable version of this matrix

Examples

```
>>> 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]])
```

conjugate()

By-element conjugation.

diff(*args)

Calculate the derivative of each element in the matrix.

Examples

```
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

See also:

[*integrate*](#) (page 499), [*limit*](#) (page 499)

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.

Examples

```
>>> 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
```

See also:

[*diofant.core.expr.Expr.equals*](#) (page 67)

integrate(*args)

Integrate each element of the matrix.

Examples

```
>>> 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]])
```

See also:

[*limit*](#) (page 499), [*diff*](#) (page 498)

limit(*args)

Calculate the limit of each element in the matrix.

Examples

```
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

See also:

[integrate](#) (page 499), [diff](#) (page 498)

4.10.5 Matrix Expressions

The Matrix expression module allows users to write down statements like

```
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).inverse()*Y
X-1*X.T-1*Y
```

```
>>> Matrix(X)
Matrix([
[X[0, 0], X[0, 1], X[0, 2]],
[X[1, 0], X[1, 1], X[1, 2]],
[X[2, 0], X[2, 1], X[2, 2]]])
```

```
>>> (X*Y)[1, 2]
X[1, 0]*Y[0, 2] + X[1, 1]*Y[1, 2] + X[1, 2]*Y[2, 2]
```

where X and Y are [MatrixSymbol](#) (page 501)'s rather than scalar symbols.

Matrix Expressions Core Reference

class diofant.matrices.expressions.**MatrixExpr**(*args, **kwargs)

Superclass for Matrix Expressions

MatrixExprs represent abstract matrices, linear transformations represented within a particular basis.

Examples

```
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).inverse() * A * y
```

See also:

[MatrixSymbol](#) (page 501), [MatAdd](#) (page 501), [MatMul](#) (page 502), [Transpose](#) (page 502), [Inverse](#) (page 502)

property T

Matrix transposition.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type ImmutableMatrix.

Examples

```
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

[as_mutable](#) (page 501)
returns mutable Matrix type

as_mutable()

Returns a dense, mutable matrix with elements represented explicitly

Examples

```
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as_mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

[as_explicit](#) (page 500)
returns ImmutableMatrix

equals(*other*)

Test elementwise equality between matrices, potentially of different types

```
>>> Identity(3).equals(eye(3))
True
```

class diofant.matrices.expressions.**MatrixSymbol**(*name, n, m, **assumptions*)

Symbolic representation of a Matrix object

Creates a Diofant Symbol to represent a Matrix. This matrix has a shape and can be included in Matrix Expressions

```
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

class diofant.matrices.expressions.**MatAdd**(*args, **kwargs)

A Sum of Matrix Expressions

MatAdd inherits from and operates like Diofant Add

```
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

class diofant.matrices.expressions.**MatMul**(*args, **kwargs)

A product of matrix expressions

Examples

```
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

class diofant.matrices.expressions.**MatPow**(base, exp)

Power of matrix expression.

class diofant.matrices.expressions.**Inverse**(mat)

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

```
>>> 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
```

class diofant.matrices.expressions.**Transpose**(*args, **kwargs)

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

```
>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)
>>> Transpose(A)
A.T
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B).T
>>> transpose(A*B)
B.T*A.T
```

class diofant.matrices.expressions.Trace(mat)

Matrix Trace

Represents the trace of a matrix expression.

```
>>> A = MatrixSymbol('A', 3, 3)
>>> Trace(A)
Trace(A)
```

class diofant.matrices.expressions.FunctionMatrix(rows, cols, lamda)

Represents a Matrix using a function (Lambda)

This class is an alternative to SparseMatrix

```
>>> 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
```

class diofant.matrices.expressions.Identity(n)

The Matrix Identity I - multiplicative identity

```
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

class diofant.matrices.expressions.ZeroMatrix(m, n)

The Matrix Zero 0 - additive identity

```
>>> A = MatrixSymbol('A', 3, 5)
>>> Z = ZeroMatrix(3, 5)
>>> A+Z
A
>>> Z*A.T
0
```

Block Matrices

Block matrices allow you to construct larger matrices out of smaller sub-blocks. They can work with [MatrixExpr](#) (page 500) or [ImmutableMatrix](#) (page 498) objects.

class diofant.matrices.expressions.blockmatrix.BlockMatrix(*args)

A BlockMatrix is a Matrix composed of other smaller, submatrices

The submatrices are stored in a Diofant Matrix object but accessed as part of a Matrix Expression

```
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> B
Matrix([
[X, Z]
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> C
Matrix([[I, Z]])
```

```
>>> block_collapse(C*B)
Matrix([[X, Z + Z*Y]])
```

transpose()

Return transpose of matrix.

Examples

```
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> B.transpose()
Matrix([
[X.T, 0]
[Z.T, Y.T]])
>>> .transpose()
Matrix([
[X, Z]
[0, Y]])
```

class diofant.matrices.expressions.blockmatrix.**BlockDiagMatrix**(*mats)

A BlockDiagMatrix is a BlockMatrix with matrices only along the diagonal

```
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> BlockDiagMatrix(X, Y)
Matrix([
[X, 0]
[0, Y]])
```

diofant.matrices.expressions.blockmatrix.**block_collapse**(expr)

Evaluates a block matrix expression

```
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> B
Matrix([
[X, Z]
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> C
Matrix([[I, Z]])
```

```
>>> block_collapse(C*B)
Matrix([[X, Z + Z*Y]])
```

4.11 Polynomials

Polynomial manipulation algorithms and algebraic objects.

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. Here we document a dedicated module 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.

4.11.1 Basic polynomial manipulation functions

User-friendly public interface to polynomial functions.

class diofant.polys.polytools.**GroebnerBasis**(*F*, **gens*, ***args*)

Represents a reduced Gröbner basis.

contains(*poly*)

Check if *poly* belongs the ideal generated by *self*.

Examples

```
>>> 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
```

property dimension

Dimension of the ideal, generated by a Gröbner basis.

property independent_sets

Compute independent sets for ideal, generated by a Gröbner basis.

References

- [KW88]

reduce(*expr*, *auto=True*)

Reduces a polynomial modulo a Gröbner basis.

Given a polynomial *f* and a set of polynomials $G = (g_1, \dots, g_n)$, computes a set of quotients $q = (q_1, \dots, q_n)$ and the remainder *r* such that $f = q_1*f_1 + \dots + q_n*f_n + r$, where *r* vanishes or *r* is a completely reduced polynomial with respect to *G*.

Examples

```
>>> 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
```

set_order(*order*)

Convert a Gröbner basis from one ordering to another.

Notes

The FGLM algorithm [FaugereGLM93] used to convert reduced Gröbner bases of zero-dimensional ideals from one ordering to another. Sometimes it is infeasible to compute a Gröbner basis with respect to a particular ordering directly.

Examples

```
>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, order='grlex')
```

```
>>> G.set_order('lex') == groebner(F, order='lex')
True
```

diofant.polys.polytools.LC(*f*, **gens*, ***args*)

Return the leading coefficient of *f*.

Examples

```
>>> LC(4*x**2 + 2*x*y**2 + x*y + 3*y)
4
```

diofant.polys.polytools.LM(*f*, **gens*, ***args*)

Return the leading monomial of *f*.

Examples

```
>>> LM(4*x**2 + 2*x*y**2 + x*y + 3*y)
x**2
```

diofant.polys.polytools.LT(*f*, **gens*, ***args*)

Return the leading term of *f*.

Examples

```
>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

class diofant.polys.polytools.**Poly**(*rep, *gens, **args*)

Generic class for representing polynomial expressions.

EC(*order=None*)

Returns the last non-zero coefficient of *self*.

Examples

```
>>> (x**3 + 2*x**2 + 3*x).as_poly().EC()
3
```

EM(*order=None*)

Returns the last non-zero monomial of *self*.

Examples

```
>>> (4*x**2 + 2*x*y**2 + x*y + 3*y).as_poly().EM()
x**0*y**1
```

ET(*order=None*)

Returns the last non-zero term of *self*.

Examples

```
>>> (4*x**2 + 2*x*y**2 + x*y + 3*y).as_poly().ET()
(x**0*y**1, 3)
```

LC(*order=None*)

Returns the leading coefficient of *self*.

Examples

```
>>> (4*x**3 + 2*x**2 + 3*x).as_poly().LC()
4
```

LM(*order=None*)

Returns the leading monomial of *self*.

The leading monomial signifies the the monomial having the highest power of the principal generator in the polynomial expression.

Examples

```
>>> (4*x**2 + 2*x*y**2 + x*y + 3*y).as_poly().LM()  
x**2*y**0
```

LT(*order=None*)

Returns the leading term of *self*.

The leading term signifies the term having the highest power of the principal generator in the polynomial expression.

Examples

```
>>> (4*x**2 + 2*x*y**2 + x*y + 3*y).as_poly().LT()  
(x**2*y**0, 4)
```

TC()

Returns the trailing coefficient of *self*.

Examples

```
>>> (x**3 + 2*x**2 + 3*x).as_poly().TC()  
0
```

all_coeffs()

Returns all coefficients from a univariate polynomial *self*.

Examples

```
>>> (x**3 + 2*x - 1).as_poly().all_coeffs()  
[-1, 2, 0, 1]
```

all_roots(*multiple=True*, *radicals=True*)

Return a list of real and complex roots with multiplicities.

Examples

```
>>> (2*x**3 - 7*x**2 + 4*x + 4).as_poly().all_roots()  
[-1/2, 2, 2]  
>>> (x**3 + x + 1).as_poly().all_roots()  
[RootOf(x**3 + x + 1, 0), RootOf(x**3 + x + 1, 1),  
 RootOf(x**3 + x + 1, 2)]
```

property args

Don't mess up with the core.

Examples

```
>>> (x**2 + 1).as_poly().args
(x**2 + 1, x)
```

as_dict(*native=False*)

Switch to a **dict** representation.

Examples

```
>>> (x**2 + 2*x*y**2 - y).as_poly().as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}
```

as_expr(**gens*)

Convert a Poly instance to an Expr instance.

Examples

```
>>> f = (x**2 + 2*x*y**2 - y).as_poly()
```

```
>>> 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
```

cancel(*other, include=False*)

Cancel common factors in a rational function *self/other*.

Examples

```
>>> (2*x**2 - 2).as_poly().cancel((x**2 - 2*x + 1).as_poly())
(1, Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

```
>>> (2*x**2 - 2).as_poly().cancel((x**2 - 2*x + 1).as_poly(), include=True)
(Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

clear_denoms(*convert=False*)

Clear denominators, but keep the ground domain.

Examples

```
>>> f = (x/2 + Rational(1, 3)).as_poly()
```

```
>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))
>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

coeff_monomial(*monom*)

Returns the coefficient of *monom* in *self* if there, else None.

Examples

```
>>> p = (24*x*y*exp(8) + 23*x).as_poly(greedy=False)
```

```
>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*E**8
>>> p.coeff_monomial((1, 1))
24*E**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*E**8*y + 23
>>> p.as_expr().coeff(y)
24*E**8*x
>>> p.as_expr().coeff(x*y)
24*E**8
```

coeffs(*order=None*)

Returns all non-zero coefficients from `self` in lex order.

Examples

```
>>> (x**3 + 2*x + 3).as_poly().coeffs()
[1, 2, 3]
```

See also:

[*all_coeffs*](#) (page 508), [*coeff_monomial*](#) (page 509)

cofactors(*other*)

Returns the GCD of `self` and `other` and their cofactors.

For two polynomials `f` and `g` it 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

```
>>> (x**2 - 1).as_poly().cofactors((x**2 - 3*x + 2).as_poly())
(Poly(x - 1, x, domain='ZZ'),
 Poly(x + 1, x, domain='ZZ'),
 Poly(x - 2, x, domain='ZZ'))
```

compose(*other*)

Computes the functional composition of `self` and `other`.

Examples

```
>>> (x**2 + x).as_poly().compose((x - 1).as_poly())
Poly(x**2 - x, x, domain='ZZ')
```

content()

Returns the GCD of polynomial coefficients.

Examples

```
>>> (6*x**2 + 8*x + 12).as_poly().content()
2
```

count_roots(*inf=None*, *sup=None*)

Return the number of roots of self in [*inf*, *sup*] interval.

Examples

```
>>> (x**4 - 4).as_poly().count_roots(-3, 3)
2
>>> (x**4 - 4).as_poly().count_roots(0, 1 + 3*I)
1
```

decompose()

Computes a functional decomposition of self.

Examples

```
>>> (x**4 + 2*x**3 - x - 1).as_poly().decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]
```

degree(*gen=0*)

Returns degree of self in *x_j*.

The degree of 0 is negative floating-point infinity.

Examples

```
>>> (x**2 + y*x + 1).as_poly().degree()
2
>>> (x**2 + y*x + y).as_poly().degree(y)
1
>>> Integer(0).as_poly(x).degree()
-inf
```

discriminant()

Computes the discriminant of self.

Examples

```
>>> (x**2 + 2*x + 3).as_poly().discriminant()
-8
```

dispersionset(*other=None*)

Compute the *dispersion set* of two polynomials.

Examples

```
>>> ((x - 3)*(x + 3)).as_poly().dispersionset()
{0, 6}
```

div(*other, auto=True*)

Polynomial division with remainder of self by other.

Examples

```
>>> (x**2 + 1).as_poly().div((2*x - 4).as_poly())
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))
```

```
>>> (x**2 + 1).as_poly().div((2*x - 4).as_poly(), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

property domain

Get the ground domain of self.

drop(**gens*)

Drop selected generators, if possible.

Examples

```
>>> f = (x + 1).as_poly(x, y)
```

```
>>> f.drop(y)
Poly(x + 1, x, domain='ZZ')
>>> f.drop(x)
Traceback (most recent call last):
ValueError: can't drop (Symbol('x'),)
```

eject(**gens*)

Eject selected generators into the ground domain.

Examples

```
>>> f = (x**2*y + x*y**3 + x*y + 1).as_poly()
```

```
>>> f.eject(x)
Poly(x*y**3 + (x**2 + x)*y + 1, y, domain='ZZ[x]')
>>> f.eject(y)
Poly(y*x**2 + (y**3 + y)*x + 1, x, domain='ZZ[y]')
```

eval(*x*, *a=None*, *auto=True*)

Evaluate self at a in the given variable.

Examples

```
>>> (x**2 + 2*x + 3).as_poly().eval(2)
11
```

```
>>> (2*x*y + 3*x + y + 2).as_poly().eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')
```

```
>>> f = (2*x*y + 3*x + y + 2*z).as_poly()
```

```
>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45
```

```
>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')
>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')
```

exclude()

Remove unnecessary generators from self.

Examples

```
>>> (a + x).as_poly(a, b, c, d, x).exclude()
Poly(a + x, a, x, domain='ZZ')
```

exquo(*other*, *auto=True*)

Computes polynomial exact quotient of self by other.

Examples

```
>>> (x**2 - 1).as_poly().exquo((x - 1).as_poly())
Poly(x + 1, x, domain='ZZ')
```

```
>>> (x**2 + 1).as_poly().exquo((2*x - 4).as_poly())
Traceback (most recent call last):
ExactQuotientFailedError: 2*x - 4 does not divide x**2 + 1
```

exquo_ground(*coeff*)

Exact quotient of self by a an element of the ground domain.

Examples

```
>>> (2*x + 4).as_poly().exquo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> (2*x + 3).as_poly().exquo_ground(2)
Traceback (most recent call last):
ExactQuotientFailedError: 2 does not divide 3 in ZZ
```

factor_list()

Returns a list of irreducible factors of self.

Examples

```
>>> f = (2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y).as_poly()
```

```
>>> f.factor_list()
(2, [(Poly(x + y, x, y, domain='ZZ'), 1),
      (Poly(x**2 + 1, x, y, domain='ZZ'), 2)])
```

property free_symbols

Free symbols of a polynomial expression.

Examples

```
>>> (x**2 + 1).as_poly().free_symbols
{x}
>>> (x**2 + y).as_poly().free_symbols
{x, y}
>>> (x**2 + y).as_poly(x).free_symbols
{x, y}
```

property free_symbols_in_domain

Free symbols of the domain of self.

Examples

```
>>> (x**2 + 1).as_poly().free_symbols_in_domain
set()
>>> (x**2 + y).as_poly().free_symbols_in_domain
set()
>>> (x**2 + y).as_poly(x).free_symbols_in_domain
{y}
```

classmethod from_dict(*rep*, **gens*, ***args*)

Construct a polynomial from a **dict**.

classmethod from_expr(*rep*, **gens*, ***args*)

Construct a polynomial from an expression.

classmethod from_list(*rep*, **gens*, ***args*)

Construct a polynomial from a **list**.

classmethod from_poly(*rep*, **gens*, ***args*)

Construct a polynomial from a polynomial.

gcd(*other*)

Returns the polynomial GCD of self and other.

Examples

```
>>> (x**2 - 1).as_poly().gcd((x**2 - 3*x + 2).as_poly())
Poly(x - 1, x, domain='ZZ')
```

gcdex(*other*, *auto*=True)

Extended Euclidean algorithm of self and other.

Returns (*s*, *t*, *h*) such that $h = \text{gcd}(f, g)$ and $s*f + t*g = h$.

Examples

```
>>> f = (x**4 - 2*x**3 - 6*x**2 + 12*x + 15).as_poly()
>>> g = (x**3 + x**2 - 4*x - 4).as_poly()
```

```
>>> f.gcdex(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'))
```

property gen

Return the principal generator.

Examples

```
>>> (x**2 + 1).as_poly().gen  
x
```

get_modulus()

Get the modulus of self.

Examples

```
>>> (x**2 + 1).as_poly(modulus=2).get_modulus()  
2
```

half_gcdex(other, auto=True)

Half extended Euclidean algorithm of self and other.

Returns (s, h) such that $h = \gcd(f, g)$ and $s*f = h \pmod{g}$.

Examples

```
>>> f = (x**4 - 2*x**3 - 6*x**2 + 12*x + 15).as_poly()  
>>> g = (x**3 + x**2 - 4*x - 4).as_poly()
```

```
>>> f.half_gcdex(g)  
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```

has_only_gens(*gens)

Return True if `Poly(f, *gens)` retains ground domain.

Examples

```
>>> (x*y + 1).as_poly(x, y, z).has_only_gens(x, y)  
True  
>>> (x*y + z).as_poly(x, y, z).has_only_gens(x, y)  
False
```

inject(front=False)

Inject ground domain generators into self.

Examples

```
>>> f = (x**2*y + x*y**3 + x*y + 1).as_poly(x)
```

```
>>> f.inject()  
Poly(x**2*y + x*y**3 + x*y + 1, x, y, domain='ZZ')  
>>> f.inject(front=True)  
Poly(y**3*x + y*x**2 + y*x + 1, y, x, domain='ZZ')
```

integrate(*specs, **args)

Computes indefinite integral of self.

Examples

```
>>> (x**2 + 2*x + 1).as_poly().integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')
```

```
>>> (x*y**2 + x).as_poly().integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

invert(other, auto=True)

Invert self modulo other when possible.

Examples

```
>>> (x**2 - 1).as_poly().invert((2*x - 1).as_poly())
Poly(-4/3, x, domain='QQ')
```

```
>>> (x**2 - 1).as_poly().invert((x - 1).as_poly())
Traceback (most recent call last):
NotInvertibleError: zero divisor
```

property is_cyclotomic

Returns True if self is a cyclotomic polynomial.

Examples

```
>>> f = (x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1).as_poly()
>>> f.is_cyclotomic
False
```

```
>>> g = (x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1).as_poly()
>>> g.is_cyclotomic
True
```

property is_ground

Returns True if self is an element of the ground domain.

Examples

```
>>> x.as_poly().is_ground
False
>>> Integer(2).as_poly(x).is_ground
True
>>> y.as_poly(x).is_ground
True
```

property is_homogeneous

Returns True if self is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree.

Examples

```
>>> (x**2 + x*y).as_poly().is_homogeneous
True
>>> (x**3 + x*y).as_poly().is_homogeneous
False
```

property `is_irreducible`

Returns True if self has no factors over its domain.

Examples

```
>>> (x**2 + x + 1).as_poly(modulus=2).is_irreducible
True
>>> (x**2 + 1).as_poly(modulus=2).is_irreducible
False
```

property `is_linear`

Returns True if self is linear in all its variables.

Examples

```
>>> (x + y + 2).as_poly().is_linear
True
>>> (x*y + 2).as_poly().is_linear
False
```

property `is_multivariate`

Returns True if self is a multivariate polynomial.

Examples

```
>>> (x**2 + x + 1).as_poly().is_multivariate
False
>>> (x*y**2 + x*y + 1).as_poly().is_multivariate
True
>>> (x*y**2 + x*y + 1).as_poly(x).is_multivariate
False
>>> (x**2 + x + 1).as_poly(x, y).is_multivariate
True
```

property `is_one`

Returns True if self is a unit polynomial.

Examples

```
>>> Integer(0).as_poly(x).is_one
False
>>> Integer(1).as_poly(x).is_one
True
```

property `is_quadratic`

Returns True if self is quadratic in all its variables.

Examples

```
>>> (x*y + 2).as_poly().is_quadratic
True
>>> (x*y**2 + 2).as_poly().is_quadratic
False
```

property `is_squarefree`

Returns True if self is a square-free polynomial.

Examples

```
>>> (x**2 - 2*x + 1).as_poly().is_squarefree
False
>>> (x**2 - 1).as_poly().is_squarefree
True
```

property `is_term`

Returns True if self is zero or has only one term.

Examples

```
>>> (3*x**2).as_poly().is_term
True
>>> (3*x**2 + 1).as_poly().is_term
False
```

property `is_univariate`

Returns True if self is a univariate polynomial.

Examples

```
>>> (x**2 + x + 1).as_poly().is_univariate
True
>>> (x*y**2 + x*y + 1).as_poly().is_univariate
False
>>> (x*y**2 + x*y + 1).as_poly(x).is_univariate
True
>>> (x**2 + x + 1).as_poly(x, y).is_univariate
False
```

property `is_zero`

Returns True if self is a zero polynomial.

Examples

```
>>> Integer(0).as_poly(x).is_zero
True
>>> Integer(1).as_poly(x).is_zero
False
```

`lcm(other)`

Returns polynomial LCM of self and other.

Examples

```
>>> (x**2 - 1).as_poly().lcm((x**2 - 3*x + 2).as_poly())
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')
```

`length()`

Returns the number of non-zero terms in `self`.

Examples

```
>>> (x**2 + 2*x - 1).as_poly().length()
3
```

`monic(auto=True)`

Divides all coefficients by `LC(f)`.

Examples

```
>>> (3*x**2 + 6*x + 9).as_poly().monic()
Poly(x**2 + 2*x + 3, x, domain='QQ')
```

```
>>> (3*x**2 + 4*x + 2).as_poly().monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')
```

`monoms(order=None)`

Returns all non-zero monomials from `self` in lex order.

Examples

```
>>> (x**2 + 2*x*y**2 + x*y + 3*y).as_poly().monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]
```

`classmethod new(rep, *gens)`

Construct *Poly* (page 507) instance from raw representation.

`roots(n=15, maxsteps=50, cleanup=True)`

Compute numerical approximations of roots of `self`.

Parameters

- **n ... the number of digits to calculate**
- **maxsteps ... the maximum number of iterations to do**
- **If the accuracy `n` cannot be reached in `maxsteps`, it will raise an exception. You need to rerun with higher `maxsteps`.**

Examples

```
>>> (x**2 - 3).as_poly().nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> (x**2 - 3).as_poly().nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

per(*rep*, **gens*, *remove=None*)

Create a Poly out of the given representation.

Examples

```
>>> a = (x**2 + 1).as_poly()
>>> R = ZZ.inject(x)
```

```
>>> a.per(R.from_list([ZZ(1), ZZ(1)]), y)
Poly(y + 1, y, domain='ZZ')
```

primitive()

Returns the content and a primitive form of self.

Examples

```
>>> (2*x**2 + 8*x + 12).as_poly().primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

quo(*other*, *auto=True*)

Computes polynomial quotient of self by other.

Examples

```
>>> (x**2 + 1).as_poly().quo((2*x - 4).as_poly())
Poly(1/2*x + 1, x, domain='QQ')
```

```
>>> (x**2 - 1).as_poly().quo((x - 1).as_poly())
Poly(x + 1, x, domain='ZZ')
```

quo_ground(*coeff*)

Quotient of self by a an element of the ground domain.

Examples

```
>>> (2*x + 4).as_poly().quo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> (2*x + 3).as_poly().quo_ground(2)
Poly(x + 1, x, domain='ZZ')
```

rat_clear_denoms(*other*)

Clear denominators in a rational function self/other.

Examples

```
>>> f = (x**2/y + 1).as_poly(x)
>>> g = (x**3 + y).as_poly(x)
```

```
>>> p, q = f.rat_clear_denoms(g)
```

```
>>> p
Poly(x**2 + y, x, domain='ZZ[y]')
>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')
```

real_roots(*multiple=True, radicals=True*)

Return a list of real roots with multiplicities.

Examples

```
>>> (2*x**3 - 7*x**2 + 4*x + 4).as_poly().real_roots()
[-1/2, 2, 2]
>>> (x**3 + x + 1).as_poly().real_roots()
[RootOf(x**3 + x + 1, 0)]
```

rem(*other, auto=True*)

Computes the polynomial remainder of self by other.

Examples

```
>>> (x**2 + 1).as_poly().rem((2*x - 4).as_poly())
Poly(5, x, domain='ZZ')
```

```
>>> (x**2 + 1).as_poly().rem((2*x - 4).as_poly(), auto=False)
Poly(x**2 + 1, x, domain='ZZ')
```

reorder(**gens, **args*)

Efficiently apply new order of generators.

Examples

```
>>> (x**2 + x*y**2).as_poly().reorder(y, x)
Poly(y**2*x + x**2, y, x, domain='ZZ')
```

replace(*x, y=None*)

Replace x with y in generators list.

Examples

```
>>> (x**2 + 1).as_poly().replace(x, y)
Poly(y**2 + 1, y, domain='ZZ')
```

resultant(*other*, *includePRS=False*)

Computes the resultant of *self* and *other* 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 535) separately.

Examples

```
>>> f = (x**2 + 1).as_poly()
```

```
>>> f.resultant((x**2 - 1).as_poly())
4
>>> f.resultant((x**2 - 1).as_poly(), includePRS=True)
(4, [Poly(x**2 + 1, x, domain='ZZ'), Poly(x**2 - 1, x, domain='ZZ'),
Poly(-2, x, domain='ZZ')])
```

retract(*field=None*)

Recalculate the ground domain of a polynomial.

Examples

```
>>> f = (x**2 + 1).as_poly(domain=QQ.inject(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')
```

root(*index*, *radicals=True*)

Get an indexed root of a polynomial.

Examples

```
>>> f = (2*x**3 - 7*x**2 + 4*x + 4).as_poly()
```

```
>>> 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
```

```
>>> (x**5 + x + 1).as_poly().root(0)
Root0f(x**3 - x**2 + 1, 0)
```

set_domain(*domain*)

Set the ground domain of self.

set_modulus(*modulus*)

Set the modulus of self.

Examples

```
>>> (5*x**2 + 2*x - 1).as_poly().set_modulus(2)
Poly(x**2 + 1, x, modulus=2)
```

shift(*a*)Efficiently compute Taylor shift $f(x + a)$.

Examples

```
>>> (x**2 - 2*x + 1).as_poly().shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')
```

sqf_list()

Returns a list of square-free factors of self.

Examples

```
>>> f = (2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16).as_poly()
```

```
>>> f.sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])
```

sqf_norm()

Computes square-free norm of self.

Returns s, f, r , such that $g(x) = f(x-sa)$ and $r(x) = \text{Norm}(g(x))$ is a square-free polynomial over K , where a is the algebraic extension of the ground domain.

Examples

```
>>> s, f, r = (x**2 + 1).as_poly(extension=[sqrt(3)]).sqf_norm()
```

```
>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

sqf_part()

Computes square-free part of self.

Examples

```
>>> (x**3 - 3*x - 2).as_poly().sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

subresultants(*other*)

Computes the subresultant PRS of *self* and *other*.

Examples

```
>>> (x**2 + 1).as_poly().subresultants((x**2 - 1).as_poly())
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

terms(*order=None*)

Returns all non-zero terms from *self* in lex order.

Examples

```
>>> (x**2 + 2*x*y**2 + x*y + 3*y).as_poly().terms()
[((2, 0), 1), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

terms_gcd()

Remove GCD of terms from the polynomial *self*.

Examples

```
>>> (x**6*y**2 + x**3*y).as_poly().terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

termwise(*func*, **gens*, ***args*)

Apply a function to all terms of *self*.

Examples

```
>>> def func(k, coeff):
...     k = k[0]
...     return coeff//10**(2-k)
```

```
>>> (x**2 + 20*x + 400).as_poly().termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

to_exact()

Make the ground domain exact.

Examples

```
>>> (x**2 + 1.0).as_poly().to_exact()
Poly(x**2 + 1, x, domain='QQ')
```

to_field()

Make the ground domain a field.

Examples

```
>>> (x**2 + 1).as_poly().to_field()
Poly(x**2 + 1, x, domain='QQ')
```

to_ring()

Make the ground domain a ring.

Examples

```
>>> (x**2 + 1).as_poly(field=True).to_ring()
Poly(x**2 + 1, x, domain='ZZ')
```

total_degree()

Returns the total degree of self.

Examples

```
>>> (x**2 + y*x + 1).as_poly().total_degree()
2
>>> (x + y**5).as_poly().total_degree()
5
```

trunc(p)

Reduce self modulo a constant p.

Examples

```
>>> (2*x**3 + 3*x**2 + 5*x + 7).as_poly().trunc(3)
Poly(-x**3 - x + 1, x, domain='ZZ')
```

unify(other)

Make self and other belong to the same domain.

Examples

```
>>> f, g = (x/2 + 1).as_poly(), (2*x + 1).as_poly()
```

```
>>> 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')
```

class diofant.polys.polytools.**PurePoly**(*rep, *gens, **args*)

Class for representing pure polynomials.

property free_symbols

Free symbols of a polynomial.

Examples

```
>>> PurePoly(x**2 + 1).free_symbols
set()
>>> PurePoly(x**2 + y).free_symbols
set()
>>> PurePoly(x**2 + y, x).free_symbols
{y}
```

diofant.polys.polytools.cancel(*f, *gens, **args*)

Cancel common factors in a rational function *f*.

Examples

```
>>> 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
```

diofant.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 $cff = \text{quo}(f, h)$ and $cfg = \text{quo}(g, h)$ are, so called, cofactors of *f* and *g*.

Examples

```
>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

`diofant.polys.polytools.compose(f, g, *gens, **args)`
 Compute functional composition $f(g)$.

Examples

```
>>> compose(x**2 + x, x - 1)
x**2 - x
```

`diofant.polys.polytools.content(f, *gens, **args)`
 Compute GCD of coefficients of f .

Examples

```
>>> content(6*x**2 + 8*x + 12)
2
```

`diofant.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

```
>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1
```

`diofant.polys.polytools.decompose(f, *gens, **args)`
 Compute functional decomposition of f .

Examples

```
>>> decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]
```

`diofant.polys.polytools.degree(f, *gens, **args)`
 Return the degree of f in the given variable.
 The degree of 0 is negative infinity.

Examples

```
>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-inf
```

`diofant.polys.polytools.discriminant(f, *gens, **args)`
 Compute discriminant of f.

Examples

```
>>> discriminant(x**2 + 2*x + 3)
-8
```

`diofant.polys.polytools.div(f, g, *gens, **args)`
 Compute polynomial division of f and g.

Examples

```
>>> div(x**2 + 1, 2*x - 4, field=False)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4)
(x/2 + 1, 5)
```

`diofant.polys.polytools.eliminate(F, G, *gens, **args)`
 Eliminate the symbols G from the polynomials F.

Parameters

- **F** (*iterable of Expr's*) – The system of polynomials to eliminate variables from.
- **G** (*iterable of Symbol's*) – Symbols to be eliminated.

Returns

list – Equations, which are equivalent to the F, but do not contain symbols G.

Examples

```
>>> eliminate([x + y + z, y - z], [y])
[x + 2*z]
>>> eliminate([x + y + z, y - z, x - y], [y, z])
[x]
>>> eliminate([x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1], [z])
[x**2 - x - y**2 + y, 2*x*y**2 + y**4 - y**2, y**6 - 4*y**4 + 4*y**3 - y**2]
```

`diofant.polys.polytools.exquo(f, g, *gens, **args)`
 Compute polynomial exact quotient of f and g.

Examples

```
>>> exquo(x**2 - 1, x - 1)
x + 1
```

```
>>> exquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
ExactQuotientFailedError: 2*x - 4 does not divide x**2 + 1
```

`diofant.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 507) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, `factor()` (page 530) will traverse the expression tree and factor its components without any prior expansion, unless an instance of *Add* (page 104) is encountered (in this case formal factorization is used). This way `factor()` (page 530) 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`.

Examples

```
>>> 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)
```

See also:

`diofant.ntheory.factor_.factorint` (page 238)

`diofant.polys.polytools.factor_list(f, *gens, **args)`

Compute a list of irreducible factors of *f*.

Examples

```
>>> 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)])
```

`diofant.polys.polytools.gcd(f, g, *gens, **args)`

Compute GCD of f and g.

Examples

```
>>> gcd(x**2 - 1, x**2 - 3*x + 2)
x - 1
```

`diofant.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

```
>>> 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)
```

`diofant.polys.polytools.groebner(F, *gens, **args)`

Computes the reduced Gröbner basis for a set of polynomials.

Parameters

- **F** (*list*) – a set of polynomials
- ***gens** (*tuple*) – polynomial generators
- ****args** (*dict*) – a dictionary of parameters, namely

order

[str, optional] Monomial order, defaults to lex.

method

[{'buchberger', 'f5b'}, optional] Set algorithm to compute Gröbner basis. By default, an improved implementation of the Buchberger algorithm is used.

field

[bool, optional] Force coefficients domain to be a field. Defaults to False.

Examples

```
>>> F = [x*y - 2*x, 2*x**2 - y**2]
```

```
>>> groebner(F)
GroebnerBasis([2*x**2 - y**2, x*y - 2*x, y**3 - 2*y**2],
               x, y, domain='ZZ', order='lex')
```

```
>>> groebner(F, order=grevlex)
GroebnerBasis([y**3 - 2*y**2, 2*x**2 - y**2, x*y - 2*x],
               x, y, domain='ZZ', order='grevlex')
```

```
>>> groebner(F, field=True)
GroebnerBasis([x**2 - y**2/2, x*y - 2*x, y**3 - 2*y**2],
               x, y, domain='QQ', order='lex')
```

References

- [Buc01]
- [CLOShea15]

See also:

[*diofant.solvers.polysys.solve_poly_system*](#) (page 609)

`diofant.polys.polytools.half_gcdex(f, g, *gens, **args)`

Half extended Euclidean algorithm of f and g .

Returns (s, h) such that $h = \gcd(f, g)$ and $s*f = h \pmod{g}$.

Examples

```
>>> 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)
```

`diofant.polys.polytools.invert(f, g, *gens, **args)`

Invert f modulo g when possible.

Examples

```
>>> invert(x**2 - 1, 2*x - 1)
-4/3
```

```
>>> invert(x**2 - 1, x - 1)
Traceback (most recent call last):
NotInvertibleError: zero divisor
```

For more efficient inversion of Rationals, use the `mod_inverse` function:

```
>>> mod_inverse(3, 5)
2
>>> (Integer(2)/5).invert(Integer(7)/3)
5/2
```


See also:[diofant.core.numbers.mod_inverse](#) (page 91)`diofant.polys.polytools.lcm(f, g, *gens, **args)`

Compute LCM of f and g.

Examples

```
>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

`diofant.polys.polytools.monic(f, *gens, **args)`

Divide all coefficients of f by LC(f).

Examples

```
>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3
```

`diofant.polys.polytools.nroots(f, n=15, maxsteps=50, cleanup=True)`

Compute numerical approximations of roots of f.

Examples

```
>>> nroots(x**2 - 3, n=15)
[-1.73205080756888, 1.73205080756888]
>>> nroots(x**2 - 3, n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

`diofant.polys.polytools.parallel_poly_from_expr(exprs, *gens, **args)`

Construct polynomials from expressions.

`diofant.polys.polytools.primitive(f, *gens, **args)`

Compute content and the primitive form of f.

Examples

```
>>> 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)
```

`diofant.polys.polytools.quo(f, g, *gens, **args)`

Compute polynomial quotient of f and g.

Examples

```
>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1
```

`diofant.polys.polytools.real_roots(f, multiple=True)`

Return a list of real roots with multiplicities of f.

Examples

```
>>> real_roots(2*x**3 - 7*x**2 + 4*x + 4)
[-1/2, 2, 2]
```

`diofant.polys.polytools.reduced(f, G, *gens, **args)`

Reduces a polynomial f modulo a set of polynomials G.

Given a polynomial f and a set of polynomials $G = (g_1, \dots, g_n)$, computes a set of quotients $q = (q_1, \dots, q_n)$ and the remainder r such that $f = q_1 g_1 + \dots + q_n g_n + r$, where r vanishes or r is a completely reduced polynomial with respect to G.

Examples

```
>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

`diofant.polys.polytools.rem(f, g, *gens, **args)`

Compute polynomial remainder of f and g.

Examples

```
>>> rem(x**2 + 1, 2*x - 4, field=False)
x**2 + 1
>>> rem(x**2 + 1, 2*x - 4)
5
```

`diofant.polys.polytools.resultant(f, g, *gens, **args)`

Compute resultant of f and g.

Examples

```
>>> resultant(x**2 + 1, x**2 - 1)
4
```

`diofant.polys.polytools.sqf(f, *gens, **args)`

Compute square-free factorization of f.

Examples

```
>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

`diofant.polys.polytools.sqf_list(f, *gens, **args)`

Compute a list of square-free factors of f.

Examples

```
>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])
```

`diofant.polys.polytools.sqf_norm(f, *gens, **args)`

Compute square-free norm of f.

Returns s, f, r, such that $g(x) = f(x-sa)$ and $r(x) = \text{Norm}(g(x))$ is a square-free polynomial over K, where a is the algebraic extension of the ground domain.

Examples

```
>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)
```

`diofant.polys.polytools.sqf_part(f, *gens, **args)`

Compute square-free part of f.

Examples

```
>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2
```

`diofant.polys.polytools.subresultants(f, g, *gens, **args)`

Compute subresultant PRS of f and g.

Examples

```
>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

`diofant.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.

Examples

```
>>> terms_gcd(x**6*y**2 + x**3*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
```

See also:

[`diofant.core.exprtools.gcd_terms`](#) (page 141), [`diofant.core.exprtools.factor_terms`](#) (page 141)

`diofant.polys.polytools.trunc(f, p, *gens, **args)`
 Reduce f modulo a constant p .

Examples

```
>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1
```

4.11.2 Extra polynomial manipulation functions

High-level polynomials manipulation functions.

`diofant.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.

Examples

```
>>> from diofant.abc import 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))
```

References

- https://en.wikipedia.org/wiki/Horner_scheme

`diofant.polys.polyfuncs.interpolate(data, x)`
 Construct an interpolating polynomial for the data points.

Examples

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
```

`diofant.polys.polyfuncs.symmetrize(F, *gens, **args)`

Rewrite a polynomial in terms of elementary symmetric polynomials.

A symmetric polynomial is a multivariate polynomial that remains invariant under any variable permutation, i.e., if $f = f(x_1, x_2, \dots, x_n)$, then $f = f(x_{\{i_1\}}, x_{\{i_2\}}, \dots, x_{\{i_n\}})$, where (i_1, i_2, \dots, i_n) is a permutation of $(1, 2, \dots, n)$ (an element of the group S_n).

Returns a tuple of symmetric polynomials (f_1, f_2, \dots, f_n) such that $f = f_1 + f_2 + \dots + f_n$.

Examples

```
>>> 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)])
```

`diofant.polys.polyfuncs.viete(f, *gens, roots=None, **args)`

Generate Viète's formulas for f .

Examples

```
>>> viete(a*x**2 + b*x + c, x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

4.11.3 Domain constructors

Tools for constructing domains for expressions.

`diofant.polys.constructor.construct_domain(obj, **args)`

Construct a minimal domain for the list of coefficients.

4.11.4 Algebraic number fields

Computational algebraic field theory.

`diofant.polys.numberfields.field_isomorphism(a, b, **args)`

Construct an isomorphism between two number fields.

`diofant.polys.numberfields.minimal_polynomial(ex, method=None, **args)`

Computes the minimal polynomial of an algebraic element.

Parameters

- **ex** (*algebraic element expression*)
- **method** (*str, optional*) - If `compose`, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If `groebner`, a bottom-up algorithm, using Gröbner bases is used. Default value is determined by `setup()` (page 41).
- **domain** (*Domain, optional*) - If no ground domain is given, it will be generated automatically from the expression.

Examples

```
>>> minimal_polynomial(sqrt(2))(x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), domain=QQ.algebraic_field(sqrt(2)))(x)
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3))(x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0][x])(x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y))(x)
x**2 - y
```

`diofant.polys.numberfields.primitive_element(extension, **args)`

Construct a common number field for all extensions.

References

- [\[YNT89\]](#)
- [\[ARW96\]](#)

4.11.5 Monomials encoded as tuples

Tools and arithmetics for monomials of distributed polynomials.

class diofant.polys.monomials.**Monomial**(*monom*, *gens*=())

Class representing a monomial, i.e. a product of powers.

as_expr(**gens*)

Convert a monomial instance to a Diofant expression.

divides(*other*)

Check if self divides other.

gcd(*other*)

Greatest common divisor of monomials.

lcm(*other*)

Least common multiple of monomials.

4.11.6 Orderings of monomials

Definitions of monomial orderings.

class diofant.polys.orderings.**GradedLexOrder**

Graded lexicographic order of monomials.

class diofant.polys.orderings.**LexOrder**

Lexicographic order of monomials.

class diofant.polys.orderings.**ReversedGradedLexOrder**

Reversed graded lexicographic order of monomials.

4.11.7 Formal manipulation of roots of polynomials

Implementation of RootOf class and related tools.

class diofant.polys.rootoftools.**RootOf**(*f*, *x*, *index*=None, *radicals*=True,
 evaluate=None, *domain*=None,
 modulus=None)

Represents k-th root of a univariate polynomial.

The ordering used for indexing takes real roots to come before complex ones, sort complex roots by real part, then by imaginary part and finally takes complex conjugate pairs of roots to be adjacent.

Parameters

- **f** (*Expr*) – Univariate polynomial expression.

- **x** (*Symbol or Integer*) – Polynomial variable or the index of the root.
- **index** (*Integer or None, optional*) – Index of the root. If None (default), parameter x is used instead as index.
- **radicals** (*bool, optional*) – Explicitly solve linear or quadratic polynomial equation (enabled by default).
- **evaluate** (*bool or None, optional*) – Control automatic evaluation.

Examples

```
>>> expand_func(RootOf(x**3 + I*x + 2, 0))
RootOf(x**6 + 4*x**3 + x**2 + 4, 1)
```

classmethod all_roots(poly, radicals=True)

Get real and complex roots of a polynomial.

eval_rational(tol)

Returns a Rational approximation to self with the tolerance tol.

The returned instance will be at most 'tol' from the exact root.

The following example first obtains Rational approximation to 1e-7 accuracy for all roots of the 4-th order Legendre polynomial, and then evaluates it to 5 decimal digits (so all digits will be correct including rounding):

```
>>> p = legendre_poly(4, x, polys=True)
>>> roots = [r.eval_rational(Rational(1, 10)**7) for r in p.real_roots()]
>>> roots = [str(r.evalf(5)) for r in roots]
>>> roots
['-0.86114', '-0.33998', '0.33998', '0.86114']
```

property interval

Return isolation interval for the root.

classmethod real_roots(poly, radicals=True)

Get real roots of a polynomial.

refine()

Refine isolation interval for the root.

class diofant.polys.rootoftools.**RootSum**(expr, func=None, x=None, auto=True, quadratic=False)

Represents a sum of all roots of a univariate polynomial.

classmethod new(poly, func, auto=True)

Construct new RootSum instance.

4.11.8 Symbolic root-finding algorithms

Algorithms for computing symbolic roots of polynomials.

`diofant.polys.polyroots.roots(f, *gens, **flags)`

Computes symbolic roots of a univariate polynomial.

Given a univariate polynomial f with symbolic coefficients (or a list of the polynomial's coefficients), returns a dictionary with its roots and their multiplicities.

Only roots expressible via radicals will be returned. To get a complete set of roots use `RootOf` class or numerical methods instead. By default cubic and quartic formulas are used in the algorithm. To disable them because of unreadable output set `cubics=False` or `quartics=False` respectively. If cubic roots are real but are expressed in terms of complex numbers (casus irreducibilis) the `trig` flag can be set to `True` to have the solutions returned in terms of cosine and inverse cosine functions.

To get roots from a specific domain set the `filter` flag with one of the following specifiers: `Z`, `Q`, `R`, `I`, `C`. By default all roots are returned (this is equivalent to setting `filter='C'`).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the `multiple` flag to `True`; the list will have identical roots appearing next to each other in the result. (For a given `Poly`, the `all_roots` method will give the roots in sorted numerical order.)

Examples

```
>>> roots(x**2 - 1, x)
{-1: 1, 1: 1}
```

```
>>> p = (x**2 - 1).as_poly()
>>> roots(p)
{-1: 1, 1: 1}
```

```
>>> p = (x**2 - y).as_poly()
```

```
>>> roots(p.as_poly(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}
```

References

- https://en.wikipedia.org/wiki/Cubic_equation#Trigonometric_and_hyperbolic_solutions

4.11.9 Special polynomials

Functions for generating interesting polynomials, e.g. for benchmarking.

`diofant.polys.specialpolys.cyclotomic_poly(n, x=None, **args)`

Generates cyclotomic polynomial of order n in x .

`diofant.polys.specialpolys.interpolating_poly(n, x, X='x', Y='y')`

Construct Lagrange interpolating polynomial for n data points.

`diofant.polys.specialpolys.random_poly(x, n, inf, sup, domain=PythonIntegerRing(), polys=False, percent=None)`

Return a polynomial of degree n with coefficients in $[inf, sup]$.

`diofant.polys.specialpolys.swinnerton_dyer_poly(n, x=None, **args)`

Generates n -th Swinnerton-Dyer polynomial in x .

`diofant.polys.specialpolys.symmetric_poly(n, *gens, **args)`

Generates symmetric polynomial of order n .

4.11.10 Orthogonal polynomials

Efficient functions for generating orthogonal polynomials.

`diofant.polys.orthopolys.chebyshevt_poly(n, x=None, **args)`

Generates Chebyshev polynomial of the first kind of degree n in x .

`diofant.polys.orthopolys.chebyshevu_poly(n, x=None, **args)`

Generates Chebyshev polynomial of the second kind of degree n in x .

`diofant.polys.orthopolys.gegenbauer_poly(n, a, x=None, **args)`

Generates Gegenbauer polynomial of degree n in x .

`diofant.polys.orthopolys.hermite_poly(n, x=None, **args)`

Generates Hermite polynomial of degree n in x .

`diofant.polys.orthopolys.jacobi_poly(n, a, b, x=None, **args)`

Generates Jacobi polynomial of degree n in x .

`diofant.polys.orthopolys.laguerre_poly(n, x=None, alpha=None, **args)`

Generates Laguerre polynomial of degree n in x .

`diofant.polys.orthopolys.legendre_poly(n, x=None, **args)`

Generates Legendre polynomial of degree n in x .

`diofant.polys.orthopolys.spherical_bessel_fn(n, x=None, **args)`

Coefficients for the spherical Bessel functions.

Those are only needed in the `jn()` function.

The coefficients are calculated from:

$$fn(0, z) = 1/z \quad fn(1, z) = 1/z^{**2} \quad fn(n-1, z) + fn(n+1, z) == (2*n+1)/z * fn(n, z)$$

Examples

```
>>> spherical_bessel_fn(1, z)
z**(-2)
>>> spherical_bessel_fn(2, z)
-1/z + 3/z**3
>>> spherical_bessel_fn(3, z)
-6/z**2 + 15/z**4
>>> spherical_bessel_fn(4, z)
1/z - 45/z**3 + 105/z**5
```

4.11.11 Manipulation of rational functions

Tools for manipulation of rational expressions.

`diofant.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\(\)](#) (page 527). However, [together\(\)](#) (page 544) 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 544) is a complement to [apart\(\)](#) (page 545), so `apart(together(expr))` should return `expr` unchanged. Note however, that [together\(\)](#) (page 544) uses only symbolic methods, so it might be necessary to use [cancel\(\)](#) (page 527) to perform algebraic simplification and minimise degree of the numerator and denominator.

Examples

```
>>> 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))
E**(1/y + 1/x)
>>> together(exp(1/x + 1/y), deep=True)
E**((x + y)/(x*y))
```

```
>>> together(1/exp(x) + 1/(x*exp(x)))
E**(-x)*(x + 1)/x
```

```
>>> together(1/exp(2*x) + 1/(x*exp(3*x)))
E**(-3*x)*(E**x*x + 1)/x
```

4.11.12 Partial fraction decomposition

Algorithms for partial fraction decomposition of rational functions.

`diofant.polys.partfrac.apart(f, x=None, full=False, **options)`

Compute partial fraction decomposition of a rational function.

Given a rational function `f`, computes the partial fraction decomposition of `f`. Two algorithms are available: One is based on the undertermined coefficients method, the other is Bronstein's full partial fraction decomposition algorithm.

The undetermined coefficients method (selected by `full=False`) uses polynomial factorization (and therefore accepts the same options as `factor`) for the denominator. Per default it works over the rational numbers, therefore decomposition of denominators with non-rational roots (e.g. irrational, complex roots) is not supported by default (see options of `factor`).

Bronstein's algorithm can be selected by using `full=True` and allows a decomposition of denominators with non-rational roots. A human-readable result can be obtained via `doit()` (see examples below).

Examples

By default, using the undetermined coefficients method:

```
>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)
```

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*y*_a/3 - y/3)/(x - _a)))
```

Calling `doit()` yields a human-readable result:

```
>>> apart(y/(x**2 + x + 1), x, full=True).doit()
(-y/3 - 2*y*(-1/2 - sqrt(3)*I/2)/3)/(x + 1/2 + sqrt(3)*I/2) + (-y/3 -
 2*y*(-1/2 + sqrt(3)*I/2)/3)/(x + 1/2 - sqrt(3)*I/2)
```

See also:

[apart_list](#) (page 546), [assemble_partfrac_list](#) (page 547)

`diofant.polys.partfrac.apart_list(f, x=None, dummies=None, **options)`

Compute partial fraction decomposition of a rational function and return the result in structured form.

Given a rational function `f` compute the partial fraction decomposition of `f`. Only Bronstein's full partial fraction decomposition algorithm is supported by this method. The return value is highly structured and perfectly suited for further algorithmic treatment rather than being human-readable. The function returns a tuple holding three elements:

- The first item is the common coefficient, free of the variable x used for decomposition. (It is an element of the base field K .)
- The second item is the polynomial part of the decomposition. This can be the zero polynomial. (It is an element of $K[x]$.)
- The third part itself is a list of quadruples. Each quadruple has the following elements in this order:
 - The (not necessarily irreducible) polynomial D whose roots w_i appear in the linear denominator of a bunch of related fraction terms. (This item can also be a list of explicit roots. However, at the moment `apart_list` never returns a result this way, but the related `assemble_partfrac_list` function accepts this format as input.)
 - The numerator of the fraction, written as a function of the root w
 - The linear denominator of the fraction *excluding its power exponent*, written as a function of the root w .
 - The power to which the denominator has to be raised.

One can always rebuild a plain expression by using the function `assemble_partfrac_list`.

Examples

A first example:

```
>>> 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, x - _a), 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, x - _a), 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*t*_a/(4*t - 1) - t/(4*t - 1)),
  Lambda(_a, x - _a), 1)])
```

```
>>> assemble_partfrac_list(pfd)
RootSum(t + _w**2 + _w, Lambda(_a, (-2*t*_a/(4*t - 1) - t/(4*t - 1))/(x - _a)))
```

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, x - _a), 1),
  (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, x - _a), 2),
  (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, x - _a), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

See also:

[apart](#) (page 545), [assemble_partfrac_list](#) (page 547)

References

- [BS93]

`diofant.polys.partfrac.assemble_partfrac_list(partial_list)`

Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.

Examples

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, x - _a), 1),
  (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, x - _a), 2),
  (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, x - _a), 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, x - _a),
  1)])
```

```
>>> pfda = assemble_partfrac_list(pfd)
>>> pfda
RootSum(_w**2 - 2, Lambda(_a, _a/(x - _a)))/2
```

```
>>> pfda.doit()
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

```
>>> a = Dummy('a')
>>> pfd = (1, Integer(0).as_poly(x),
...       [(sqrt(2), -sqrt(2)),
...        Lambda(a, a/2), Lambda(a, -a + x), 1)])
```

```
>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

See also:

[apart](#) (page 545), [apart_list](#) (page 546)

4.12 Printing

See the [Printing](#) (page 10) section in Tutorial for introduction into printing.

This guide documents the printing system in Diofant and how it works internally.

4.12.1 Printer Class

Printing subsystem driver

Diofant'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 diofant. Every class had its own latex, mathml, str and repr methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefore all printing code was combined into the different printers, which works great for built-in diofant 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 `diofant.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

Here we will overload `StrPrinter`.

```
>>> from diofant.printing.str import StrPrinter

>>> class CustomStrPrinter(StrPrinter):
...     def _print_Derivative(self, expr): # noga: N802
...         return str(expr.args[0].func) + "*" * len(expr.args[1:])
>>> def mystr(e):
...     return CustomStrPrinter().doprint(e)
>>> print(mystr(f(t).diff((t, 2))))
f''
```

printmethod: `str` | `None` = `None`

_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.

doprint(*expr*)

Returns printer's representation for *expr* (as a string).

classmethod set_global_settings(***settings*)

Set system-wide printing settings.

4.12.2 PrettyPrinter Class

The pretty printing subsystem is implemented in `diofant.printing.pretty` by the `PrettyPrinter` class deriving from `Printer`. It relies on the modules `diofant.printing.stringPict`, and `diofant.printing.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).

class `diofant.printing.pretty.PrettyPrinter(settings=None)`

Printer, which converts an expression into 2D ASCII-art figure.

`diofant.printing.pretty.pprint(expr, **settings)`

`diofant.printing.pretty.pretty(expr, **settings)`

Returns a string containing the prettified form of `expr`.

For information on keyword arguments see `pretty_print` function.

`diofant.printing.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** (*int or None, optional*) – number of columns before line breaking (default to `None` which reads the terminal width), useful when using Diofant without terminal.
- **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`

4.12.3 CCodePrinter

This class implements C code printing (i.e. it converts Python expressions to strings of C code).

Usage:

```
>>> 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))
```

class `diofant.printing.ccode.CCodePrinter(settings={})`

A printer to convert python expressions to strings of c code.

```
printmethod: str | None = '_ccode'
```

```
indent_code(code)
```

Accepts a string of code or a list of code lines.

```
diofant.printing.ccode.ccode(expr, assign_to=None, **settings)
```

Converts an expr to a string of c code

Parameters

- **expr** (*Expr*) – A diofant expression to be converted.
- **assign_to** (*optional*) – When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.
- **precision** (*integer, optional*) – The precision for numbers such as pi [default=15].
- **user_functions** (*dict, optional*) – A dictionary where the keys are string representations of either `FunctionClass` or `UndefinedFunction` instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)] or [(argument_test, cfunction_formatter)]. See below for examples.
- **dereference** (*iterable, optional*) – An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.
- **human** (*bool, optional*) – If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].
- **contract** (*bool, optional*) – If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> ccode((2*x)**Rational(7, 2))
'8*sqrt(2)*pow(x, 7.0L/2.0L)'
>>> ccode(sin(x), assign_to='s')
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {'ceiling': 'CEIL',
...                     'Abs': [(lambda x: not x.is_integer, 'fabs'),
...                               (lambda x: x.is_integer, 'ABS')],
...                     'func': 'f'}
>>> func = Function('func')
```

(continues on next page)

(continued from previous page)

```
>>> ccode(func(abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the C-function takes a subset of the original arguments:

```
>>> ccode(2**x + 3**x,
...       user_functions={'Pow': [(lambda b, e: b == 2,
...                               lambda b, e: f'exp2({e})'),
...                               (lambda b, e: b != 2, 'pow')]}})
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, y))
if (x > 0) {
y = x + 1;
}
else {
y = x;
}
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> len_y = 5
>>> y = IndexedBase('y', shape=[len_y])
>>> t = IndexedBase('t', shape=[len_y])
>>> Dy = IndexedBase('Dy', shape=[len_y - 1])
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(ccode(mat, A))
A[0] = pow(x, 2);
if (x > 0) {
    A[1] = x + 1;
}
else {
    A[1] = x;
}
A[2] = sin(x);
```

4.12.4 Fortran Printing

The `fcode` function translates a diofant 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.

`diofant.printing.fcode.fcode(expr, assign_to=None, **settings)`

Converts an `expr` to a string of fortran code

Parameters

- **expr** (*Expr*) - A diofant expression to be converted.
- **assign_to** (*optional*) - When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.
- **precision** (*integer, optional*) - The precision for numbers such as `pi` [default=15].
- **user_functions** (*dict, optional*) - A dictionary where keys are `Function-Class` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(`argument_test`, `cfunction_string`)]. See below for examples.
- **human** (*bool, optional*) - If `True`, the result is a single string that may contain some constant declarations for the number symbols. If `False`, the same information is returned in a tuple of (`symbols_to_declare`, `not_supported_functions`, `code_text`). [default=`True`].
- **contract** (*bool, optional*) - If `True`, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=`True`].
- **source_format** (*optional*) - The source format can be either 'fixed' or 'free'. [default='fixed']
- **standard** (*integer, optional*) - The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

Examples

```
>>> fcode((2*x)**Rational(7, 2))
'      8*sqrt(2.0d0)*x**(7.0d0/2.0d0)'
>>> fcode(sin(x), assign_to='s')
's = sin(x)'
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {'ceiling': 'CEIL',
...                    'floor': [(lambda x: not x.is_integer, 'FLOOR1'),
...                              (lambda x: x.is_integer, 'FLOOR2')]}
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
'      CEIL(x) + FLOOR1(x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, y))
      if (x > 0) then
        y = x + 1
      else
        y = x
      end if
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> len_y = 5
>>> y = IndexedBase('y', shape=[len_y])
>>> t = IndexedBase('t', shape=[len_y])
>>> Dy = IndexedBase('Dy', shape=[len_y - 1])
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
'Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i))'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
      A(1, 1) = x**2
      if (x > 0) then
        A(2, 1) = x + 1
      else
        A(2, 1) = x
      end if
      A(3, 1) = sin(x)
```

```
class diofant.printing.fcode.FCodePrinter(settings={})
```

A printer to convert diofant expressions to strings of Fortran code.

printmethod: `str` | `None` = `'_fcode'`

indent_code(code)

Accepts a string of code or a list of code lines.

Two basic examples:

```
>>> 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).remove0()
>>> print(fcode(expr))
-715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
@ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
@ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
@ /2.0d0*x**2 + 1
```

In case of line wrapping, it is handy to include the assignment so that lines are wrapped properly when the assignment part is added.

```
>>> print(fcode(expr, assign_to='var'))
var = -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
@ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
@ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
@ /2.0d0*x**2 + 1
```

For piecewise functions, the `assign_to` option is mandatory:

```
>>> print(fcode(Piecewise((x, x < 1), (x**2, True)), assign_to='var'))
      if (x < 1) then
        var = x
      else
        var = x**2
      end if
```

Note that by default only top-level piecewise functions are supported due to the lack of a conditional operator in Fortran 77. Inline conditionals can be supported using the `merge` function introduced in Fortran 95 by setting of the kwarg `standard=95`:

```
>>> print(fcode(Piecewise((x, x < 1), (x**2, True)), standard=95))
      merge(x, x**2, x < 1)
```

Loops are generated if there are Indexed objects in the expression. This also requires use of the `assign_to` option.

```
>>> A, B = map(IndexedBase, ['A', 'B'])
>>> m = Symbol('m', integer=True)
>>> i = Idx('i', m)
>>> print(fcode(2*B[i], assign_to=A[i]))
      do i = 1, m
        A(i) = 2*B(i)
      end do
```

Repeated indices in an expression with Indexed objects are interpreted as summation. For instance, code for the trace of a matrix can be generated with

```
>>> print(fcode(A[i, i], assign_to=x))
      x = 0
      do i = 1, m
        x = x + A(i, i)
      end do
```

By default, number symbols such as `pi` and `E` are detected and defined as Fortran parameters. The precision of the constants can be tuned with the `precision` argument. Parameter definitions are easily avoided using the `N` function.


```
>>> print(fcode(x - pi**2 - E))
      parameter (E = 2.71828182845905d0)
      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 supported in Fortran:
C      gamma
      -gamma(x)**2 + 1
```

By default the output is human readable code, ready for copy and paste. With the option `human=False`, the return value is suitable for post-processing with source code generators that write routines with multiple instructions. The return value is a three-tuple containing: (i) a set of number symbols that must be defined as ‘Fortran parameters’, (ii) a list functions that cannot be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```
>>> fcode(1 - gamma(x)**2, human=False)
(set(), {gamma(x)}, '-gamma(x)**2 + 1')
>>> fcode(1 - sin(x)**2, human=False)
(set(), set(), '-sin(x)**2 + 1')
>>> fcode(x - pi**2, human=False)
(({pi, '3.14159265358979d0'}), set(), '      x - pi**2')
```

4.12.5 Mathematica code printing

class `diofant.printing.mathematica.MCodePrinter(settings={})`

A printer to convert python expressions to strings of the Wolfram’s Mathematica code.

printmethod: `str | None = '_mcode'`

doprint(*expr*)

Returns printer’s representation for *expr* (as a string).

`diofant.printing.mathematica.mathematica_code(expr, **settings)`

Converts an *expr* to a string of the Wolfram Mathematica code

Examples

```
>>> mathematica_code(sin(x).series(x).remove0())
'(1/120)*x^5 - 1/6*x^3 + x'
```

4.12.6 LambdaPrinter

This classes implements printing to strings that can be used by the `diofant.utilities.lambdify.lambdify()` (page 745) function.

class `diofant.printing.lambdarepr.LambdaPrinter(settings=None)`

This printer converts expressions into strings that can be used by `lambdify`.

printmethod: `str` | `None` = `'_diofantstr'`

`diofant.printing.lambdarepr.lambdarepr(expr, **settings)`

Returns a string usable for `lambdifying`.

4.12.7 LatexPrinter

This class implements LaTeX printing. See `diofant.printing.latex`.

`diofant.printing.latex.accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', 'sinh', 'cosh', 'tanh', 'sqrt', 'ln', 'log', 'sec', 'csc', 'cot', 'coth', 're', 'im', 'frac', 'root', 'arg']`

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

class `diofant.printing.latex.LatexPrinter(settings=None)`

LaTeX printer.

printmethod: `str` | `None` = `'_latex'`

`diofant.printing.latex.latex(expr, **settings)`

Convert the given expression to LaTeX representation.

```
>>> from diofant.abc import mu, r, tau
```

```
>>> print(latex((2*tau)**Rational(7, 2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

Not using a `print` statement for printing, results in double backslashes for latex commands since that's the way Python escapes backslashes in strings.

```
>>> 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”) and “none”. This parameter does nothing for Mul objects. 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 “ $\wedge\{p/q\}$ ” instead of “ $\wedge\{\frac{p}{q}\}$ ” for fractional powers.

```
>>> print(latex((2*tau)**Rational(7, 2), fold_frac_powers=True))
8 \sqrt{2} \tau^{\frac{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”, “matrix”, “array”, etc. Defaults to “smallmatrix” for inline mode, “matrix” for matrices of no more than 10 columns, and “array” otherwise.

```
>>> print(latex(Matrix(2, 1, [x, y])))
\left[ \begin{matrix} x \\ y \end{matrix} \right]
```

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_str='array'))
\left[ \begin{array}{c} x \\ y \end{array} \right]
```

`mat_delim`: The delimiter to wrap around matrices. Can be one of “[”, “(”, or the empty string. Defaults to “[”.

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_delim='('))
\left( \begin{matrix} x \\ y \end{matrix} \right)
```

`symbol_names`: Dictionary of symbols and the custom strings they should be emitted as.

```
>>> print(latex(x**2, symbol_names={x: 'x_i'}))
x_i^2
```

latex also supports the builtin container types list, tuple, and dictionary.

```
>>> print(latex([2/x, y], mode='inline'))
$\left [ 2 / x, \quad y \right ]$
```

4.12.8 MathMLPrinter

This class is responsible for MathML printing. See `diofant.printing.mathml`.

More info on mathml content: <http://www.w3.org/TR/MathML2/chapter4.html>

class `diofant.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

- <https://www.w3.org/TR/MathML3/>

printmethod: `str | None = '_mathml'`

doprint(*expr*)

Prints the expression as MathML.

mathml_tag(*e*)

Returns the MathML tag for an expression.

`diofant.printing.mathml.mathml`(*expr, **settings*)

Returns the MathML representation of *expr*.

4.12.9 PythonPrinter

This class implements Python printing. Usage:

```
>>> print(python(5*x**3 + sin(x)))
x = Symbol('x')
e = 5*x**3 + sin(x)
```

4.12.10 ReprPrinter

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

class diofant.printing.repr.ReprPrinter(*settings=None*)

Repr printer.

printmethod: `str` | `None` = `'_diofantrepr'`

emptyPrinter(*expr*)

The fallback printer.

reprify(*args*, *sep*)

Prints each item in *args* and joins them with *sep*.

diofant.printing.repr.srepr(*expr*, ***settings*)

Return *expr* in repr form.

4.12.11 StrPrinter

This module generates readable representations of Diofant expressions.

class diofant.printing.str.StrPrinter(*settings=None*)

Str printer.

printmethod: `str` | `None` = `'_diofantstr'`

diofant.printing.str.sstr(*expr*, ***settings*)

Returns the expression as a string.

For large expressions where speed is a concern, use the setting `order='none'`.

Examples

```
>>> sstr(Eq(a + b, 0))
'Eq(a + b, 0)'
```

4.12.12 Implementation - Helper Classes/Functions

`diofant.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.

```
>>> 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 `diofant.printing.codeprinter.CodePrinter(settings=None)`

The base class for code-printing subclasses.

printmethod: `str | None = '_diofantstr'`

exception `diofant.printing.codeprinter.AssignmentError`

Raised if an assignment variable for a loop is missing.

Precedence

A module providing information about the necessity of brackets

4.12.13 Pretty-Printing Implementation Helpers

`diofant.printing.pretty_symbology.U(name)`

Unicode character by name or None if not found.

The following two functions return the Unicode version of the inputted Greek letter.

`diofant.printing.pretty_symbology.g(l)`

`diofant.printing.pretty_symbology.G(l)`

```
diofant.printing.pretty_symbology.greek_letters = ['alpha', 'beta', 'gamma',
'delta', 'epsilon', 'zeta', 'eta', 'theta', 'iota', 'kappa', 'lamda', 'mu',
'nu', 'xi', 'omicron', 'pi', 'rho', 'sigma', 'tau', 'upsilon', 'phi', 'chi',
'psi', 'omega']
```

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

The following functions return Unicode vertical objects.

`diofant.printing.pretty_symbology.xobj(symb, length)`

Construct spatial object of given length.

return: [] of equal-length strings

`diofant.printing.pretty_symbology.vobj(symb, height)`

Construct vertical object of a given height.

See also:

[*xobj*](#) (page 562)

`diofant.printing.pretty_symbology.hobj(symb, width)`

Construct horizontal object of a given width.

See also:

[*xobj*](#) (page 562)

The following functions are for rendering atoms and symbols.

`diofant.printing.pretty_symbology.xsym(sym)`

Get symbology for a ‘character’.

`diofant.printing.pretty_symbology.pretty_atom(atom_name, default=None)`

Return pretty representation of an atom.

`diofant.printing.pretty_symbology.pretty_symbol(symb_name)`

Return pretty representation of a symbol.

`diofant.printing.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 `diofant.printing.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

```
>>> 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.

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

```
>>> print(stringPict('10').right(' + ', stringPict('1\r-\r2', 1))[0])
10 + 1
      2
```

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 diofant.printing.stringpict.prettyForm(s, baseline=0, binding=0)

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)

```

4.12.14 dotprint

```

diofant.printing.dot.dotprint(expr, styles=[(<class 'diofant.core.basic.Basic'>,
                                             {'color': 'blue', 'shape': 'ellipse'}), (<class
                                             'diofant.core.expr.Expr'>, {'color': 'black'})],
                               atom=<function <lambda>>, maxdepth=None,
                               repeat=True, labelfunc=<class 'str'>, **kwargs)

```

DOT description of a Diofant 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 `repr`. For example with `str`, the leaf nodes of $x + 1$ are labeled, x and 1 . With `repr`, they are labeled `Symbol('x')` and `Integer(1)`.

Additional keyword arguments are included as styles for the graph.

Examples

```

>>> print(dotprint(x + 2))
digraph{
# Graph style
"bgcolor"="transparent"
"ordering"="out"
"rankdir"="TD"
#####
# Nodes #
#####
"Add(Symbol('x'), Integer(2))_()" ["color"="black", "label"="Add", "shape"=

```

(continues on next page)

(continued from previous page)

```

->"ellipse"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];
#####
# Edges #
#####
"Add(Symbol('x'), Integer(2))_()" -> "Integer(2)_(0,)";
"Add(Symbol('x'), Integer(2))_(1)" -> "Symbol('x')_(1,)";
}

```

4.13 Interactive

Helper module for setting up interactive Diofant sessions.

AST/string-based transformations, provided here, [could be used](#) in IPython to reduce boilerplate while interacting with Diofant due to the Python language syntax.

```
diofant.interactive.printing.init_printing(no_global=False, pretty_print=None,
                                             **settings)
```

Initializes pretty-printer depending on the environment.

Parameters

- **no_global** (*boolean*) - If True, the settings become system wide; if False, use just for this console/session.
- **pretty_print** (*boolean or None*) - Enable pretty printer (turned on by default for IPython, but disabled for plain Python console).
- ****settings** (*dict*) - A dictionary of default settings for printers.

Notes

This function runs automatically for wildcard imports (e.g. for `from diofant import *`) in interactive sessions.

Examples

```

>>> from diofant.abc import theta
>>> sqrt(5)
sqrt(5)
>>> init_printing(pretty_print=True, no_global=True)
>>> sqrt(5)
 $\sqrt{5}$ 
>>> theta
 $\theta$ 
>>> init_printing(pretty_print=True, order='grevlex', no_global=True)
>>> y + x + y**2 + x**2
 $x^2 + y^2 + x + y$ 

```

```
class diofant.interactive.session.AutomaticSymbols(ns={})
```

Add missing [Symbol](#) (page 80) definitions.

```
class diofant.interactive.session.FloatRationalizer
```

Wraps all floats in a call to [Fraction](#).

class diofant.interactive.session.IntegerDivisionWrapper

Wrap all int divisions in a call to [Fraction](#).

diofant.interactive.session.unicode_identifiers(*lines*)

Transform original code to allow any unicode identifiers.

diofant.interactive.session.wrap_float_literals(*lines*)

Wraps all float literals with [Float](#) (page 85).

4.14 Sets

Generic set theory interfaces.

class diofant.sets.sets.Set(*args)

The base class for any kind of set.

This is not meant to be used directly as a container of items. It does not behave like the builtin set; see [FiniteSet](#) (page 575) for that.

Real intervals are represented by the [Interval](#) (page 572) class and unions of sets by the [Union](#) (page 575) class. The empty set is represented by the [EmptySet](#) (page 578) class and available as a singleton as `S.EmptySet`.

property boundary

The boundary or frontier of a set

A point x is on the boundary of a set S if

1. x is in the closure of S . I.e. Every neighborhood of x contains a point in S .
2. x is not in the interior of S . I.e. There does not exist an open set centered on x contained entirely within S .

There are the points on the outer rim of S . If S is open then these points need not actually be contained within S .

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

Examples

```
>>> Interval(0, 1).boundary
{0, 1}
>>> Interval(0, 1, True, False).boundary
{0, 1}
```

property closure

Return the closure of a set.

Examples

```
>>> Interval(0, 1, right_open=True).closure
[0, 1]
```

complement(*universe*)

The complement of ‘self’ w.r.t the given the universe.

Examples

```
>>> Interval(0, 1).complement(S.Reals)
(-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:

Examples

```
>>> Interval(0, 1).contains(0.5)
true
>>> 0.5 in Interval(0, 1)
True
```

property inf

The infimum of ‘self’

Examples

```
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

property interior

Return the interior of a set.

The interior of a set consists all points of a set that do not belong to its boundary.

Examples

```
>>> Interval(0, 1).interior
(0, 1)
>>> Interval(0, 1).boundary.interior
EmptySet()
```

intersection(*other*)

Returns the intersection of ‘self’ and ‘other’.

```
>>> Interval(1, 3).intersection(Interval(1, 2))
[1, 2]
```

property is_closed

Test if a set is closed.

Examples

```
>>> Interval(0, 1).is_closed
True
```

is_disjoint(*other*)

Returns True if ‘self’ and ‘other’ are disjoint

Examples

```
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

References

- https://en.wikipedia.org/wiki/Disjoint_sets

property is_open

Test if a set is open.

A set is open if it has an empty intersection with its boundary.

Examples

```
>>> S.Reals.is_open
True
```

See also:

boundary (page 567)

is_proper_subset(*other*)

Returns True if ‘self’ is a proper subset of ‘other’.

Examples

```
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

is_proper_superset(*other*)

Returns True if ‘self’ is a proper superset of ‘other’.

Examples

```
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

is_subset(*other*)

Returns True if ‘self’ is a subset of ‘other’.

Examples

```
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

is_superset(*other*)

Returns True if ‘self’ is a superset of ‘other’.

Examples

```
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superset(Interval(0, 1, left_open=True))
True
```

isdisjoint(*other*)

Alias for *is_disjoint()* (page 569).

issubset(*other*)

Alias for *is_subset()* (page 570).

issuperset(*other*)

Alias for *is_superset()* (page 570).

property measure

The (Lebesgue) measure of ‘self’

Examples

```
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

powerset()

Find the Power set of ‘self’.

Examples

```
>>> A = EmptySet()
>>> A.powerset()
{EmptySet()}
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet())
True
```

References

- https://en.wikipedia.org/wiki/Power_set

property sup

The supremum of ‘self’

Examples

```
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

symmetric_difference(*other*)

Returns symmetric difference of self and other.

Examples

```
>>> Interval(1, 3).symmetric_difference(Reals)
(-oo, 1) U (3, oo)
```

References

- https://en.wikipedia.org/wiki/Symmetric_difference

union(*other*)

Returns the union of ‘self’ and ‘other’.

Examples

As a shortcut it is possible to use the ‘+’ operator:

```
>>> 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) ∪ (2, 3]
```

`diofant.sets.sets.imageset(*args)`

Image of set under transformation f .

If this function can't compute the image, it returns an unevaluated ImageSet object.

$$f(x)|x \in self$$

Examples

```
>>> imageset(x, 2*x, Interval(0, 2))
[0, 4]
```

```
>>> imageset(lambda x: 2*x, Interval(0, 2))
[0, 4]
```

```
>>> imageset(Lambda(x, sin(x)), Interval(-1, 2))
[-sin(1), 1]
```

See also:

[`diofant.sets.fancysets.ImageSet`](#) (page 579)

4.14.1 Elementary Sets

class `diofant.sets.sets.Interval`(*start=-oo, end=oo, left_open=False, right_open=False*)

Represents a real interval as a Set.

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.

Examples

```
>>> Interval(0, 1)
[0, 1]
>>> Interval(0, 1, False, True)
[0, 1)
>>> Interval.Ropen(0, 1)
[0, 1)
>>> Interval.Lopen(0, 1)
(0, 1]
>>> Interval.open(0, 1)
(0, 1)
```

```
>>> a = Symbol('a', real=True)
>>> Interval(0, a)
[0, a]
```


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

- https://en.wikipedia.org/wiki/Interval_%28mathematics%29

classmethod `Lopen(a, b)`

Return an interval not including the left boundary.

classmethod `Ropen(a, b)`

Return an interval not including the right boundary.

as_relational(x)

Rewrite an interval in terms of inequalities and logic operators.

property `end`

The right end point of ‘self’.

This property takes the same value as the ‘sup’ property.

Examples

```
>>> Interval(0, 1).end
1
```

property `inf`

The left end point of ‘self’.

This property takes the same value as the ‘inf’ property.

Examples

```
>>> Interval(0, 1).start
0
```

property `is_left_unbounded`

Return True if the left endpoint is negative infinity.

property `is_right_unbounded`

Return True if the right endpoint is positive infinity.

property `left`

The left end point of ‘self’.

This property takes the same value as the ‘inf’ property.

Examples

```
>>> Interval(0, 1).start
0
```

property left_open

True if 'self' is left-open.

Examples

```
>>> Interval(0, 1, left_open=True).left_open
true
>>> Interval(0, 1, left_open=False).left_open
false
```

classmethod open(*a*, *b*)

Return an interval including neither boundary.

property right

The right end point of 'self'.

This property takes the same value as the 'sup' property.

Examples

```
>>> Interval(0, 1).end
1
```

property right_open

True if 'self' is right-open.

Examples

```
>>> Interval(0, 1, right_open=True).right_open
true
>>> Interval(0, 1, right_open=False).right_open
false
```

property start

The left end point of 'self'.

This property takes the same value as the 'inf' property.

Examples

```
>>> Interval(0, 1).start
0
```

property sup

The right end point of 'self'.

This property takes the same value as the 'sup' property.

Examples

```
>>> Interval(0, 1).end
1
```

class diofant.sets.sets.**FiniteSet**(*args, **kwargs)

Represents a set that has a finite number of elements.

Examples

```
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
>>> 3 in FiniteSet(1, 2, 3, 4)
True
```

References

- https://en.wikipedia.org/wiki/Finite_set

as_relational(symbol)

Rewrite a FiniteSet in terms of equalities and logic operators.

4.14.2 Compound Sets

class diofant.sets.sets.**Union**(*args, **kwargs)

Represents a union of sets as a [Set](#) (page 567).

Examples

```
>>> Union(Interval(1, 2), Interval(3, 4))
[1, 2] ∪ [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]
```

See also:

[Intersection](#) (page 576)

References

- https://en.wikipedia.org/wiki/Union_%28set_theory%29

as_relational(*symbol*)

Rewrite a Union in terms of equalities and logic operators.

static reduce(*args*)

Simplify a *Union* (page 575) 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

class diofant.sets.sets.**Intersection**(*args, **kwargs)

Represents an intersection of sets as a *Set* (page 567).

Examples

```
>>> Intersection(Interval(1, 3), Interval(2, 4))
[2, 3]
```

We often use the `.intersect` method

```
>>> Interval(1, 3).intersection(Interval(2, 4))
[2, 3]
```

See also:

Union (page 575)

References

- https://en.wikipedia.org/wiki/Intersection_%28set_theory%29

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

class diofant.sets.sets.**ProductSet**(*sets, **assumptions)

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.

Examples

```
>>> 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]
```

```
>>> H, T = Symbol('H'), Symbol('T')
>>> coin = FiniteSet(H, T)
>>> set(coin**2)
{(H, H), (H, T), (T, H), (T, T)}
```

Notes

- Passes most operations down to the argument sets
- Flattens Products of ProductSets

References

- https://en.wikipedia.org/wiki/Cartesian_product

class diofant.sets.sets.**Complement**(*a*, *b*, *evaluate=True*)

Represents relative complement of a set with another set.

$$A - B = \{x \in A | x \notin B\}$$

Examples

```
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
{0, 2}
```

See also:

Intersection (page 576), *Union* (page 575)

References

- <https://mathworld.wolfram.com/ComplementSet.html>

static reduce(*A*, *B*)

Simplify a *Complement* (page 577).

4.14.3 Singleton Sets

class diofant.sets.sets.**EmptySet**(*args, **kwargs)

Represents the empty set.

The empty set is available as a singleton as S.EmptySet.

Examples

```
>>> S.EmptySet
EmptySet()
```

```
>>> Interval(1, 2).intersection(S.EmptySet)
EmptySet()
```

References

- https://en.wikipedia.org/wiki/Empty_set

4.14.4 Special Sets

Special sets.

class diofant.sets.fancysets.**Naturals**(*args, **kwargs)

The set of natural numbers.

Represents the natural numbers (or counting numbers) which are all positive integers starting from 1. This set is also available as the Singleton, S.Naturals.

Examples

```
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> S.Naturals.intersection(Interval(0, 10))
Range(1, 11, 1)
```

See also:

Naturals0 (page 578)

non-negative integers

Integers (page 579)

also includes negative integers

class diofant.sets.fancysets.**Naturals0**(*args, **kwargs)

The set of natural numbers, starting from 0.

Represents the whole numbers which are all the non-negative integers, inclusive of zero.

See also:

Naturals (page 578)

positive integers

Integers (page 579)

also includes the negative integers

class diofant.sets.fancysets.**Integers**(*args, **kwargs)

The set of all integers.

Represents all integers: positive, negative and zero. This set is also available as the Singleton, S.Integers.

Examples

```
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
>>> next(iterable)
-1
>>> next(iterable)
2
```

```
>>> S.Integers.intersection(Interval(-4, 4))
Range(-4, 5, 1)
```

See also:

Naturals0 (page 578)

non-negative integers

Integers (page 579)

positive and negative integers and zero

class diofant.sets.fancysets.**Rationals**(*args, **kwargs)

The set of all rationals.

class diofant.sets.fancysets.**ImageSet**(lamda, base_set)

Image of a set under a mathematical function.

Examples

```
>>> squares = ImageSet(Lambda(x, x**2), S.Naturals)
>>> 4 in squares
True
>>> 5 in squares
False
```

```
>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersection(squares)
{1, 4, 9}
```

```
>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
4
9
16
```

If you want to get value for $x = 2, 1/2$ etc. (Please check whether the x value is in *base_set* or not before passing it as args)

```
>>> squares.lamda(2)
4
>>> squares.lamda(Rational(1, 2))
1/4
```

class diofant.sets.fancysets.**Range**(*args)

Represents a range of integers.

Examples

```
>>> list(Range(5))
[0, 1, 2, 3, 4]
>>> list(Range(10, 15))
[10, 11, 12, 13, 14]
>>> list(Range(10, 20, 2))
[10, 12, 14, 16, 18]
>>> list(Range(20, 10, -2))
[20, 18, 16, 14, 12]
```

class diofant.sets.fancysets.**ExtendedReals**(*args, **kwargs)

The set of all extended reals.

class diofant.sets.fancysets.**Reals**(*args, **kwargs)

The set of all reals.

4.15 Simplify

4.15.1 simplify

diofant.simplify.simplify.**simplify**(*expr*, *ratio*=1.7, *measure*=<function count_ops>, *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 Diofant. 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:

```
>>> 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:

```
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
x + 1
```

In some cases, applying `simplify()` (page 580) may actually result in some more complicated expression. The default `ratio=1.7` prevents more extreme cases: if $(\text{result length})/(\text{input length}) > \text{ratio}$, then input is returned unmodified. The `measure` parameter lets you specify the function used to determine how complex an expression is. The function should take a single argument as an expression and return a number such that if expression `a` is more complex than expression `b`, then `measure(a) > measure(b)`. The default measure function is `count_ops()` (page 133), which returns the total number of operations in the expression.

For example, if `ratio=1`, `simplify` output can't be longer than input.

```
>>> 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 necessarily 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:

```
>>> a, b = symbols('a b', positive=True, real=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
```

(continues on next page)

(continued from previous page)

```
>>> h
log(a*h**(-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
```

```
>>> def my_measure(expr):
...     # 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(Integer(1)))
...     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.

4.15.2 separatevars

`diofant.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 diofant.abc import alpha
>>> 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) is None
True
```

4.15.3 nthroot

diofant.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)

Notes

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.

Examples

```
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

4.15.4 besselsimp

`diofant.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 diofant.abc import nu
>>> besselsimp(besselj(nu, z*polar_lift(-1)))
E**(I*pi*nu)*besselj(nu, z)
>>> besselsimp(besseli(nu, z*polar_lift(-I)))
E**(-I*pi*nu/2)*besselj(nu, z)
>>> besselsimp(besseli(Rational(-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
```

4.15.5 hypersimp

`diofant.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.

Notes

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.

References

- W. Koepf, **Algorithms for m-fold Hypergeometric Summation**,
Journal of Symbolic Computation (1995) 20, 399-417

4.15.6 hypersimilar

`diofant.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.

See also:

[*hypersimp*](#) (page 584)

4.15.7 nsimplify

`diofant.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).

Examples

```
>>> 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])
E**(-pi/2)
>>> nsimplify(pi, tolerance=0.01)
22/7
```

See also:

[*diofant.core.function.nfloat*](#) (page 138)

4.15.8 posify

`diofant.simplify.simplify.posify(eq)`

Return `eq` (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.

Any symbol that has `positive=None` will be replaced with a positive dummy symbol having the same name. This replacement will allow more symbolic processing of expressions, especially those involving powers and logarithms.

A dictionary that can be sent to `subs` to restore `eq` to its original symbols is also returned.

```
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(n + p + _x, {_x: x})
```

```
>>> eq = 1/x
>>> log(eq).expand()
log(1/x)
>>> log(posify(eq)[0]).expand()
-log(_x)
>>> p, rep = posify(eq)
>>> log(p).expand().subs(rep)
-log(x)
```

It is possible to apply the same transformations to an iterable of expressions:

```
>>> eq = x**2 - 4
>>> solve(eq, x)
[{x: -2}, {x: 2}]
>>> eq_x, reps = posify([eq, x])
>>> eq_x
[_x**2 - 4, _x]
>>> solve(*eq_x)
[[_x: 2]]
```

4.15.9 logcombine

`diofant.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.

Examples

```
>>> 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)
```

See also:

[*posify* \(page 586\)](#)

replace all symbols with symbols having positive assumptions

4.15.10 Radsimp

radsimp

`diofant.simplify.radsimp.radsimp(expr, symbolic=True, max_terms=4)`

Rationalize the denominator by removing square roots.

Note: the expression returned from `radsimp` must be used with caution since if the denominator contains symbols, it will be possible to make substitutions that violate the assumptions of the simplification process: that for a denominator matching $a + b\sqrt{c}$, $a \neq +/b\sqrt{c}$. (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

```
>>> radsimp(1/(I + 1))
(1 - I)/2
>>> radsimp(1/(2 + sqrt(2)))
(-sqrt(2) + 2)/2
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)
```

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5))
>>> pprint(ans)

$$\frac{\sqrt{5} \cdot a + \sqrt{5} \cdot b - \sqrt{2} \cdot x - \sqrt{2} \cdot y}{5 \cdot a^2 + 10 \cdot a \cdot b + 5 \cdot b^2 - 2 \cdot x^2 - 4 \cdot x \cdot y - 2 \cdot y^2}$$

>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))

$$\frac{\sqrt{5} \cdot (a + b) - \sqrt{2} \cdot (x + y)}{5 \cdot a^2 + 10 \cdot a \cdot b + 5 \cdot b^2 - 2 \cdot x^2 - 4 \cdot x \cdot y - 2 \cdot y^2}$$

```

If radicals in the denominator cannot be removed or there is no denominator, the original expression will be returned.

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

Results with symbols will not always be valid for all substitutions:

```
>>> eq = 1/(a + b*sqrt(c))
>>> eq.subs({a: b*sqrt(c)})
1/(2*b*sqrt(c))
>>> radsimp(eq).subs({a: b*sqrt(c)})
nan
```

If symbolic=False, symbolic denominators will not be transformed (but numeric denominators will still be processed):

```
>>> radsimp(eq, symbolic=False)
1/(a + b*sqrt(c))
```

rad_rationalize

diofant.simplify.radsimp.**rad_rationalize**(num, den)

Rationalize num/den by removing square roots in the denominator; num and den are sum of terms whose squares are rationals

Examples

```
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

collect

diofant.simplify.radsimp.**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 588), so user is expected to provide an expression in an appropriate form (for example, by using `expand()` (page 129) prior to calling this function). This makes `collect()` (page 588) 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()` (page 137) function.

Parameters

- **expr** (*Expr*) – an expression
- **syms** (*iterable of Symbol's*) – collected symbols
- **evaluate** (*boolean*) – First, if evaluate flag is set (by default), this function will return an expression with collected terms else it will return a dictionary with expressions up to rational powers as keys and collected coefficients as values.

Examples

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[1]
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:

```
>>> 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 588) function apply to the exponential function, so you can get:

```
>>> collect(a*exp(2*x) + b*exp(2*x), exp(x))
E**(2*x)*(a + b)
```

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:

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

```
>>> collect(a*Derivative(f, x) + b*Derivative(f, x), Derivative(f, x))
(a + b)*Derivative(f(x), x)
```

```
>>> collect(a*Derivative(f, (x, 2)) + b*Derivative(f, (x, 2)), f)
(a + b)*Derivative(f(x), x, x)
```

```
>>> collect(a*Derivative(f, (x, 2)) + b*Derivative(f, (x, 2)),
...        Derivative(f, x), exact=True)
a*Derivative(f(x), x, x) + b*Derivative(f(x), x, x)
```

```
>>> collect(a*Derivative(f, x) + b*Derivative(f, x) + a*f + b*f, f)
f(x)*(a + b) + (a + b)*Derivative(f(x), x)
```

Or you can even match both derivative order and exponent at the same time:

```
>>> collect(a*Derivative(f, (x, 2))**2 + b*Derivative(f, (x, 2))**2,
...        Derivative(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
```

See also:

`collect_const` (page 592), `collect_sqrt` (page 591), `rcollect` (page 590)

`diofant.simplify.radsimp.rcollect(expr, *vars)`

Recursively collect sums in an expression.

Examples

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

```
>>> rcollect(expr, y)
(x + y*(x**2 + x + 1))/(x + y)
```

See also:

[collect](#) (page 588), [collect_const](#) (page 592), [collect_sqrt](#) (page 591)

collect_sqrt

`diofant.simplify.radsimp.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.

Examples

```
>>> 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)
```

See also:

[collect](#) (page 588), [collect_const](#) (page 592), [rcollect](#) (page 590)

collect_const

diofant.simplify.radsimp.**collect_const**(*expr*, **vars*, ***kwargs*)

A non-greedy collection of terms with similar number coefficients in an Add expr. If *vars* is given then only those constants will be targeted. Although any Number can also be targeted, if this is not desired set *Numbers=False* and no Float or Rational will be collected.

Examples

```

>>> from diofant.abc import s
>>> 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)

```

See also:

[collect](#) (page 588), [collect_sqrt](#) (page 591), [rcollect](#) (page 590)

fraction

diofant.simplify.radsimp.**fraction**(*expr*, *exact=False*)

Returns a pair with expression's numerator and denominator.

If the given expression is not a fraction then this function will return the tuple (*expr*, 1).

This function will not make any attempt to simplify nested fractions or to do any term rewriting at all.

If only one of the numerator/denominator pair is needed then use `numerator(expr)` or `denominator(expr)` functions respectively.

```

>>> 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:

```
>>> fraction(2*x**(-y))
(2, x**y)
```

```
>>> fraction(exp(-x))
(1, E**x)
```

```
>>> fraction(exp(-x), exact=True)
(E**(-x), 1)
```

4.15.11 Ratsimp

ratsimp

`diofant.simplify.ratsimp.ratsimp(expr)`

Put an expression over a common denominator, cancel and reduce.

Examples

```
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)
```

4.15.12 Trigonometric simplification

trigsimp

`diofant.simplify.trigsimp.trigsimp(expr, **opts)`

Reduces expression by using known trig identities.

Notes

method: - Determine the method to use. Valid choices are 'matching' (default), 'groebner', 'combined', and 'fu'. If 'matching', simplify the expression recursively by targeting common patterns. If 'groebner', apply an experimental groebner basis algorithm. In this case further options are forwarded to `trigsimp_groebner`, please refer to its docstring. If 'combined', first run the groebner basis algorithm with small default parameters, then run the 'matching' algorithm. 'fu' runs the collection of trigonometric transformations described by Fu, et al.

Examples

```
>>> 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'`.

```
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

See also:

[`diofant.simplify.fu.fu`](#) (page 595)

futrig

`diofant.simplify.trigsimp.futrig(e, **kwargs)`

Return simplified `e` using Fu-like transformations. This is not the "Fu" algorithm. This is called by default from `trigsimp`. By default, hyperbolics subexpressions will be simplified, but this can be disabled by setting `hyper=False`.

Examples

```
>>> trigsimp(1/tan(x)**2)
tan(x)**(-2)
```

```
>>> futrig(sinh(x)/tanh(x))
cosh(x)
```

fu

diofant.simplify.fu.fu(rv, measure=<function <lambda>>)

Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.

fu() (page 595) will try to minimize the objective function measure. By default this first minimizes the number of trig terms and then minimizes the number of total operations.

Examples

```
>>> fu(sin(50)**2 + cos(50)**2 + sin(pi/6))
3/2
>>> fu(sqrt(6)*cos(x) + sqrt(2)*sin(x))
2*sqrt(2)*sin(x + pi/3)
```

CTR1 example

```
>>> eq = sin(x)**4 - cos(y)**2 + sin(y)**2 + 2*cos(x)**2
>>> fu(eq)
cos(x)**4 - 2*cos(y)**2 + 2
```

CTR2 example

```
>>> fu(Rational(1, 2) - cos(2*x)/2)
sin(x)**2
```

CTR3 example

```
>>> fu(sin(a)*(cos(b) - sin(b)) + cos(a)*(sin(b) + cos(b)))
sqrt(2)*sin(a + b + pi/4)
```

CTR4 example

```
>>> fu(sqrt(3)*cos(x)/2 + sin(x)/2)
sin(x + pi/3)
```

Example 1

```
>>> fu(1-sin(2*x)**2/4-sin(y)**2-cos(x)**4)
-cos(x)**2 + cos(y)**2
```

Example 2

```
>>> fu(cos(4*pi/9))
sin(pi/18)
>>> fu(cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9))
1/16
```

Example 3

```
>>> fu(tan(7*pi/18)+tan(5*pi/18)-sqrt(3)*tan(5*pi/18)*tan(7*pi/18))
-sqrt(3)
```

Objective function example

```
>>> fu(sin(x)/cos(x)) # default objective function
tan(x)
>>> fu(sin(x)/cos(x), measure=lambda x: -x.count_ops()) # maximize op count
sin(x)/cos(x)
```

References

http://rfdz.ph-noe.ac.at/fileadmin/Mathematik_Uploads/ACDCA/TIME2006/DES_contribs/Fu/simplification.pdf

DES-

4.15.13 Power simplification

powsimp

`diofant.simplify.powsimp.powsimp(expr, deep=False, combine='all', force=False, measure=<function count_ops>)`

Reduces expression by combining powers with similar bases and exponents.

Notes

If `deep` is `True` then `powsimp()` will also simplify arguments of functions. By default `deep` is set to `False`.

If `force` is `True` then bases will be combined without checking for assumptions, e.g. `sqrt(x)*sqrt(y) -> 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:

$x^a * y^a \Rightarrow (x*y)^a$ as well as things like $2^{2x} \Rightarrow 4^x$

and `combine='exp'` will only combine

$x^a * x^b \Rightarrow x^{(a + b)}$

`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

```
>>> 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(E**x*E**y)
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> 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
sqrt(x*sqrt(y))**5
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

powdenest

diofant.simplify.powsimp.**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, $(bb1^{be1} * bb2^{be2} \dots)^{be}$, simplification can be done as follows:

- if e is positive, the gcd of all be_i 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. $\exp(3*(\log(a) + 2*\log(b))) \rightarrow a^{*3}b^{*6}$.

Examples

```
>>> powdenest((x**(2*a/3))**(3*x))
(x**(2*a/3))**(3*x)
>>> powdenest(exp(3*x*log(2)))
2**(3*x)
```

Assumptions may prevent expansion:

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

```
>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p
```

No other expansion is done.

```
>>> i, j = symbols('i j', integer=True)
>>> powdenest((x**x)**(i + j)) # -X-> (x**x)**i*(x**x)**j
x**(x*(i + j))
```

But `exp()` will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the `exp` to a power with a different base:

```
>>> powdenest(exp(3*y*log(x)))
x**(3*y)
>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y
>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3
```

If assumptions allow, symbols can also be moved to the outermost exponent:

```
>>> i = Symbol('i', integer=True)
>>> powdenest(((x**(2*i))**(3*y))**x)
((x**(2*i))**(3*y))**x
>>> powdenest(((x**(2*i))**(3*y))**x, force=True)
x**(6*i*x*y)
```

```
>>> powdenest(((x**(2*a/3))**(3*y/i))**x)
((x**(2*a/3))**(3*y/i))**x
>>> powdenest((x**(2*i)*y**(4*i))**z, force=True)
(x*y**2)**(2*i*z)
```

```
>>> n = Symbol('n', negative=True)
```

```
>>> powdenest((x**i)**y, force=True)
x**(i*y)
>>> powdenest((n**i)**x, force=True)
(n**i)**x
```

4.15.14 Combinatorial simplification

combsimp

`diofant.simplify.combsimp.combsimp(expr)`

Simplify combinatorial expressions.

This function takes as input an expression containing factorials, binomials, Pochhammer symbol and other “combinatorial” functions, and tries to minimize the number of those functions and reduce the size of their arguments.

The algorithm works by rewriting all combinatorial functions as 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 of more familiar functions. If the initial expression consisted of gamma functions alone, the result is expressed in terms of gamma functions. If the initial expression consists of gamma function with some other combinatorial, 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 $\text{gamma}(x) \cdot \text{gamma}(1-x) == \pi / \sin(\pi \cdot x)$.
2. Reduce the number of gammas by applying the multiplication theorem $\text{gamma}(x) \cdot \text{gamma}(x+1/n) \cdot \dots \cdot \text{gamma}(x+(n-1)/n) == C \cdot \text{gamma}(n \cdot x)$.
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

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

4.15.15 Square Root Denest

sqrtdenest

`diofant.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].

Examples

```
>>> sqrtidenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

See also:

[unrad](#) (page 600)

References

- <https://researcher.watson.ibm.com/researcher/files/us-fagin/symb85.pdf>
- D. J. Jeffrey and A. D. Rich, 'Simplifying Square Roots of Square Roots by Denesting' (available at <http://www.cybertester.com/data/denest.pdf>)

unrad

`diofant.simplify.sqrtidenest.unrad(eq, *syms, **flags)`

Remove radicals with symbolic arguments and return (eq, cov), None or raise an error:

None is returned if there are no radicals to remove.

NotImplementedError is raised if there are radicals and they cannot be removed or if the relationship between the original symbols and the change of variable needed to rewrite the system as a polynomial cannot be solved.

Otherwise the tuple, (eq, cov), is returned where:

```
``eq``, ``cov``
    ``eq`` is an equation without radicals (in the symbol(s) of
    interest) whose solutions are a superset of the solutions to the
    original expression. ``eq`` might be re-written in terms of a new
    variable; the relationship to the original variables is given by
    ``cov`` which is a list containing ``v`` and ``v**p - b`` where
    ``p`` is the power needed to clear the radical and ``b`` is the
    radical now expressed as a polynomial in the symbols of interest.
    For example, for sqrt(2 - x) the tuple would be
    ``(c, c**2 - 2 + x)``. The solutions of ``eq`` will contain
    solutions to the original equation (if there are any).
```

syms

an iterable of symbols which, if provided, will limit the focus of radical removal: only radicals with one or more of the symbols of interest will be cleared. All free symbols are used if syms is not set.

flags are used internally for communication during recursive calls. Two options are also recognized:

```
``take``, when defined, is interpreted as a single-argument function
that returns True if a given Pow should be handled.
```

Radicals can be removed from an expression if:

```
* all bases of the radicals are the same; a change of variables is
  done in this case.
* if all radicals appear in one term of the expression
* there are only 4 terms with sqrt() factors or there are less than
```

(continues on next page)

(continued from previous page)

```

four terms having sqrt() factors
* there are only two terms with radicals

```

Examples

```

>>> unrad(sqrt(x)*cbrt(x) + 2)
(x**5 - 64, [])
>>> unrad(sqrt(x) + root(x + 1, 3))
(x**3 - x**2 - 2*x - 1, [])
>>> eq = sqrt(x) + root(x, 3) - 2
>>> unrad(eq)
(_p**3 + _p**2 - 2, [_p, -x + _p**6])

```

4.15.16 Common Subexpresion Elimination

cse

```

diofant.simplify.cse_main.cse(exprs, symbols=None, optimizations=None,
                               postprocess=None, order='canonical', ignore=())

```

Perform common subexpression elimination on an expression.

Parameters

- **exprs** (*list of diofant expressions, or a single diofant expression*) - The expressions to reduce.
- **symbols** (*infinite iterator yielding unique Symbols*) - The symbols used to label the common subexpressions which are pulled out. The numbered_symbols generator is useful. The default is a stream of symbols of the form "x0", "x1", etc. This must be an infinite iterator.
- **optimizations** (*list of (callable, callable) pairs*) - The (preprocessor, postprocessor) pairs of external optimization functions. Optionally 'basic' can be passed for a set of predefined basic optimizations. Such 'basic' optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.
- **postprocess** (*a function which accepts the two return values of cse and*) - returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda: `lambda r, e: return reversed(r), e`
- **order** (*string, 'none' or 'canonical'*) - The order by which Mul and Add arguments are processed. If set to 'canonical', arguments will be canonically ordered. If set to 'none', ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting `order='none'`.
- **ignore** (*iterable of Symbol's*) - Substitutions containing these symbols will be ignored.

Returns

- **replacements** (*list of (Symbol, expression) pairs*) - All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.
- **reduced_exprs** (*list of diofant expressions*) - The reduced expressions with all of the replacements above.

Examples

```
>>> from diofant.abc import w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

Note that currently, $y + z$ will not get substituted if $-y - z$ is used.

```
>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols: `>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))` `([(x0, x + 1)], [x0*y**2, 3*x0*y**2])`

opt_cse

`diofant.simplify.cse_main.opt_cse(exprs, order='canonical')`

Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls

Parameters

- **exprs** (*list of diofant expressions*) - The expressions to optimize.
- **order** (*string, 'none' or 'canonical'*) - The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.

Returns

opt_subs (*dictionary of expression substitutions*) - The expression substitutions which can be useful to optimize CSE.

Examples

```
>>> opt_subs = opt_cse([x**2])
>>> opt_subs
{x**(-2): 1/(x**2)}
```

tree_cse

`diofant.simplify.cse_main.tree_cse(exprs, symbols, opt_subs={}, order='canonical', ignore=())`

Perform raw CSE on expression tree, taking `opt_subs` into account.

Parameters

- **exprs** (*list of diofant expressions*) - The expressions to reduce.
- **symbols** (*infinite iterator yielding unique Symbols*) - The symbols used to label the common subexpressions which are pulled out.
- **opt_subs** (*dictionary of expression substitutions*) - The expressions to be substituted before any CSE action is performed.
- **order** (*string, 'none' or 'canonical'*) - The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.
- **ignore** (*iterable of Symbol's*) - Substitutions containing these symbols will be ignored.

4.15.17 Hypergeometric Function Expansion

hyperexpand

`diofant.simplify.hyperexpand.hyperexpand(f, allow_hyper=False, rewrite='default', place=None)`

Expand hypergeometric functions. If `allow_hyper` is `True`, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

If a G-function has expansions both at zero and at infinity, `place` can be set to `0` or `zoo` to indicate the preferred choice.

Examples

```
>>> hyperexpand(hyper([], [], z))
E**z
```

Non-hypergeometric parts of the expression and hypergeometric expressions that are not recognised are left unchanged:

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

4.15.18 Traversal Tools

use

`diofant.simplify.traversaltools.use(expr, func, level=0, args=(), kwargs={})`

Use `func` to transform `expr` at the given level.

Examples

```
>>> 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
```

4.15.19 EPath Tools

EPath class

`class diofant.simplify.epathtools.EPath(path)`

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 also:

[`epath`](#) (page 605)

`apply(expr, func, args=None, kwargs=None)`

Modify parts of an expression selected by a path.

Examples

```
>>> 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

```
>>> 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

`diofant.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 605) 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

```
>>> 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)
```

4.16 Solvers

This module implements methods for solving equations and inequalities.

4.16.1 Algebraic equations

This module contain solvers for all kinds of equations, algebraic or transcendental.

`diofant.solvers.solvers.minsolve_linear_system(system, *symbols, **flags)`

Find a particular solution to a linear system.

In particular, try to find a solution with the minimal possible number of non-zero variables. This is a very computationally hard problem.

Parameters

- **system** (*Matrix*) - Nx(M+1) matrix, which means it has to be in augmented form.
- ***symbols** (*list*) - List of M Symbol's.
- ****flags** (*dict*) - A dictionary of following parameters:

quick

[boolean, optional] If True, a heuristic is used. Otherwise (default) a naive algorithm with exponential complexity is used.

`diofant.solvers.solvers.solve(f, *symbols, **flags)`

Algebraically solves equation or system of equations.

Parameters

- **f** (*Expr, Equality or iterable of above*) - All expressions are assumed to be equal to 0.
- ***symbols** (*tuple*) - If none symbols given (empty tuple), free symbols of expressions will be used.
- ****flags** (*dict*) - A dictionary of following parameters:

check

[bool, optional] If False, don't do any testing of solutions. Default is True, i.e. the solutions are checked and those that doesn't satisfy given assumptions on symbols solved for or make any denominator zero - are automatically excluded.

warn

[bool, optional] Show a warning if `checksol()` (page 695) could not conclude. Default is False.

simplify

[bool, optional] Enable simplification (default) for all but polynomials of order 3 or greater before returning them and (if check is not False) use the general simplify function on the solutions and the expression obtained when they are substituted into the function which should be zero.

rational

[bool or None, optional] If True, recast Floats as Rational. If None (default), 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.

cubics, quartics, quintics

[bool, optional] Return explicit solutions (with radicals, which can be quite long) when, respectively, cubic, quartic or quintic expressions are encountered. Default is True. If False, `RootOf` (page 540) instances will be returned instead.

Examples

Single equation:

```
>>> solve(x**2 - y**2)
[{x: -y}, {x: y}]
>>> solve(x**2 - 1)
[{x: -1}, {x: 1}]
```

We could restrict solutions by using assumptions:

```
>>> p = Symbol('p', positive=True)
>>> solve(p**2 - 1)
[{p: 1}]
```

Several equations:

```
>>> solve((x + 5*y - 2, -3*x + 6*y - 15))
[{x: -3, y: 1}]
>>> solve((x + 5*y - 2, -3*x + 6*y - z))
[{x: -5*z/21 + 4/7, y: z/21 + 2/7}]
```

No solution:

```
>>> solve([x + 3, x - 3])  
[]
```

Notes

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))  
[{f(x): x}]  
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))  
[{Derivative(f(x), x): x + f(x)}]
```

See also:

[diofant.solvers.recurr.rsolve](#) (page 684)

solving recurrence equations

[diofant.solvers.ode.dsolve](#) (page 635)

solving differential equations

[diofant.solvers.inequalities.reduce_inequalities](#) (page 610)

solving inequalities

`diofant.solvers.solvers.solve_linear(f, x)`

Solve equation f wrt variable x .

Returns

tuple – $(x, \text{solution})$, if there is a linear solution, $(0, 1)$ if f is independent of the symbol x , $(0, 0)$ if solution set any denominator of f to zero or (numerator, denominator) of f , if it's a nonlinear expression wrt x .

Examples

```
>>> solve_linear(1/x - y**2, x)  
(x, y**(-2))  
>>> solve_linear(x**2/y**2 - 3, x)  
(x**2 - 3*y**2, y**2)  
>>> solve_linear(y, x)  
(0, 1)  
>>> solve_linear(1/(1/x - 2), x)  
(0, 0)
```

Systems of Polynomial Equations

Solvers of systems of polynomial equations.

`diofant.solvers.polysys.solve_linear_system(system, *symbols, **flags)`

Solve system of linear equations.

Both under- and overdetermined systems are supported. The possible number of solutions is zero, one or infinite.

Parameters

- **system** (*Matrix*) - $N \times (M+1)$ matrix, which means it has to be in augmented form. This matrix will not be modified.
- ***symbols** (*list*) - List of M Symbol's

Returns

solution (*dict or None*) - 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.

Examples

Solve the following system:

$$\begin{array}{rcl} x + 4y & == & 2 \\ -2x + y & == & 14 \end{array}$$

```
>>> 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)
{}
```

See also:

[`diofant.matrices.matrices.MatrixBase.rref`](#) (page 468)

`diofant.solvers.polysys.solve_poly_system(eqs, *gens, **args)`

Solve a system of polynomial equations.

Polynomial system may have finite number of solutions or infinitely many (positive-dimensional systems).

References

- [CLOShea15], p. 98

Examples

```
>>> solve_poly_system([x*y - 2*y, 2*y**2 - x**2], x, y)
[{x: 0, y: 0}, {x: 2, y: -sqrt(2)}, {x: 2, y: sqrt(2)}]
```

```
>>> solve_poly_system([x*y], x, y)
[{x: 0}, {y: 0}]
```

`diofant.solvers.polysys.solve_surd_system(eqs, *gens, **args)`

Solve a system of algebraic equations.

Examples

```
>>> solve_surd_system([x + sqrt(x + 1) - 2])
[{x: -sqrt(13)/2 + 5/2}]
```

4.16.2 Inequality Solvers

Tools for solving inequalities and systems of inequalities.

`diofant.solvers.inequalities.reduce_inequalities(inequalities, symbols=[])`

Reduces a system of inequalities or equations.

Examples

```
>>> reduce_inequalities(0 <= x + 3, [])
-3 <= x
>>> reduce_inequalities(0 <= x + y*2 - 1, [x])
-2*y + 1 <= x
```

See also:

[`diofant.solvers.solvers.solve`](#) (page 606)

solve algebraic equations

4.16.3 Diophantine

Diophantine equations

The word “Diophantine” comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the “father of Algebra”, Diophantus in his famous work “Arithmetica” presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a “Diophantine equation” to an equation of the form, $f(x_1, x_2, \dots, x_n) = 0$ where $n \geq 2$ and x_1, x_2, \dots, x_n are integer variables. If we can find n integers a_1, a_2, \dots, a_n such that $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$ satisfies the above equation, we say that the equation is solvable.

Currently, following five types of Diophantine equations can be solved using [`diophantine\(\)`](#) (page 615) and other helper functions of the Diophantine module.

- Linear Diophantine equations: $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$.
- General binary quadratic equation: $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation: $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$
- Extended Pythagorean equation: $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
- General sum of squares: $x_1^2 + x_2^2 + \dots + x_n^2 = k$

Module structure

This module contains *diophantine()* (page 615) and helper functions that are needed to solve certain Diophantine equations. It's structured in the following manner.

- *diophantine()* (page 615)
 - *diop_solve()* (page 617)
 - * *classify_diop()* (page 616)
 - * *diop_linear()* (page 618)
 - * *diop_quadratic()* (page 619)
 - * *diop_ternary_quadratic()* (page 623)
 - * *diop_ternary_quadratic_normal()* (page 631)
 - * *diop_general_pythagorean()* (page 624)
 - * *diop_general_sum_of_squares()* (page 625)
 - * *diop_general_sum_of_even_powers()* (page 625)
 - *merge_solution()* (page 629)

When an equation is given to *diophantine()* (page 615), it factors the equation(if possible) and solves the equation given by each factor by calling *diop_solve()* (page 617) separately. Then all the results are combined using *merge_solution()* (page 629).

diop_solve() (page 617) internally uses *classify_diop()* (page 616) to find the type of the equation(and some other details) given to it and then calls the appropriate solver function based on the type returned. For example, if *classify_diop()* (page 616) returned “linear” as the type of the equation, then *diop_solve()* (page 617) calls *diop_linear()* (page 618) to solve the equation.

Each of the functions, *diop_linear()* (page 618), *diop_quadratic()* (page 619), *diop_ternary_quadratic()* (page 623), *diop_general_pythagorean()* (page 624) and *diop_general_sum_of_squares()* (page 625) 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.

Tutorial

First, let's import the highest API of the Diophantine module.

```
>>> from diofant.solvers.diophantine import diophantine
```

Before we start solving the equations, we need to define the variables.

```
>>> x, y, z, t, p, q = symbols('x, y, z, t, p, q', integer=True)
>>> t1, t2, t3, t4, t5 = symbols('t1:6', integer=True)
```

Let's start by solving the easiest type of Diophantine equations, i.e. linear Diophantine equations. Let's solve $2x + 3y = 5$. Note that although we write the equation in the above form, when we input the equation to any of the functions in Diophantine module, it needs to be in the form $eq = 0$.

```
>>> diophantine(2*x + 3*y - 5)
{(3*t_0 - 5, -2*t_0 + 5)}
```

Note that stepping one more level below the highest API, we can solve the very same equation by calling `diop_solve()` (page 617).

```
>>> from diofant.solvers.diophantine import diop_solve
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
```

Note that it returns a tuple rather than a set. `diophantine()` (page 615) always return a set of tuples. But `diop_solve()` (page 617) may return a single tuple or a set of tuples depending on the type of the equation given.

We can also solve this equation by calling `diop_linear()` (page 618), which is what `diop_solve()` (page 617) calls internally.

```
>>> from diofant.solvers.diophantine import diop_linear
>>> diop_linear(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
```

If the given equation has no solutions then the outputs will look like below.

```
>>> diophantine(2*x + 4*y - 3)
set()
>>> diop_solve(2*x + 4*y - 3)
(None, None)
>>> diop_linear(2*x + 4*y - 3)
(None, None)
```

Note that except for the highest level API, in case of no solutions, a tuple of `None` are returned. Size of the tuple is the same as the number of variables. Also, one can specifically set the parameter to be used in the solutions by passing a customized parameter. Consider the following example:

```
>>> diop_solve(2*x + 3*y - 5, m)
(3*m_0 - 5, -2*m_0 + 5)
```

For linear Diophantine equations, the customized parameter is the prefix used for each free variable in the solution. Consider the following example:

```
>>> diop_solve(2*x + 3*y - 5*z + 7, m)
(m_0, m_0 + 5*m_1 - 14, m_0 + 3*m_1 - 7)
```

In the solution above, `m_0` and `m_1` are independent free variables.

Please note that for the moment, users can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let's try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases associated with the equation would help a lot. Let us define $\Delta = b^2 - 4ac$ w.r.t. the binary quadratic $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

When $\Delta < 0$, there are either no solutions or only a finite number of solutions.

```
>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
{(2, 1), (5, 1)}
```

In the above equation $\Delta = (-4)^2 - 4 * 1 * 8 = -16$ and hence only a finite number of solutions exist.

When $\Delta = 0$ we might have either no solutions or parameterized solutions.


```
>>> diophantine(3*x**2 - 6*x*y + 3*y**2 - 3*x + 7*y - 5)
set()
>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
{(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)}
>>> diophantine(x**2 + 2*x*y + y**2 - 3*x - 3*y)
{(t_0, -t_0), (t_0, -t_0 + 3)}
```

The most interesting case is when $\Delta > 0$ and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where $\Delta = 8$.

```
>>> diophantine(x**2 - 4*x*y + 2*y**2 - 3*x + 7*y - 5)
set()
>>> s = diophantine(x**2 - 2*y**2 - 2*x - 4*y, n)
>>> x_1, y_1 = s.pop()
>>> x_2, y_2 = s.pop()
>>> x_n = (-(-2*sqrt(2) + 3)**n/2 + sqrt(2)*(-2*sqrt(2) + 3)**n/2 -
...        sqrt(2)*(2*sqrt(2) + 3)**n/2 - (2*sqrt(2) + 3)**n/2 + 1)
>>> x_1 == x_n or x_2 == x_n
True
>>> y_n = (-sqrt(2)*(-2*sqrt(2) + 3)**n/4 + (-2*sqrt(2) + 3)**n/2 +
...        sqrt(2)*(2*sqrt(2) + 3)**n/4 + (2*sqrt(2) + 3)**n/2 - 1)
>>> y_1 == y_n or y_2 == y_n
True
```

Here n is an integer. Although x_n and y_n may not look like integers, substituting in specific values for n (and simplifying) shows that they are. For example consider the following example where we set n equal to 9.

```
>>> 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 diofant.solvers.diophantine import diop_DN, find_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()` (page 621)

```
>>> 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()` (page 619)

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation has no solution.

Now let's turn to homogeneous ternary quadratic equations. These equations are of the form $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$. These type of equations either have infinitely many solutions or no solutions (except the obvious solution (0, 0, 0))

```
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
{(0, 0, 0)}
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
{(-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + 68*q**2)}
```

If you are only interested in a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use [diop_ternary_quadratic\(\)](#) (page 623).

```
>>> from diofant.solvers.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
(-4, 5, 1)
```

[diop_ternary_quadratic\(\)](#) (page 623) first converts the given equation to an equivalent equation of the form $w^2 = AX^2 + BY^2$ and then it uses [descent\(\)](#) (page 624) to solve the latter equation. You can refer to the docs of [transformation_to_normal\(\)](#) (page 635) to find more on this. The equation $w^2 = AX^2 + BY^2$ can be solved more easily by using the Aforementioned [descent\(\)](#) (page 624).

```
>>> from diofant.solvers.diophantine import descent
>>> descent(3, 1) # solves the equation w**2 = 3*Y**2 + Z**2
(1, 0, 1)
```

Here the solution tuple is in the order (w, Y, Z)

The extended Pythagorean equation, $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$ and the general sum of squares equation, $x_1^2 + x_2^2 + \dots + x_n^2 = k$ can also be solved using the Diofantine module.

```
>>> from diofant.abc import e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
{(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5,
 60*t3*t5, 210*t4*t5, 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)}
```

function [diop_general_pythagorean\(\)](#) (page 624) can also be called directly to solve the same equation. Either you can call [diop_general_pythagorean\(\)](#) (page 624) or use the high level API. For the general sum of squares, this is also true, but one advantage of calling [diop_general_sum_of_squares\(\)](#) (page 625) is that you can control how many solutions are returned.

```
>>> from diofant.solvers.diophantine import diop_general_sum_of_squares
>>> eq = a**2 + b**2 + c**2 + d**2 - 18
>>> diophantine(eq)
{(0, 0, 3, 3), (0, 1, 1, 4), (1, 2, 2, 3)}
>>> diop_general_sum_of_squares(eq, 2)
{(0, 0, 3, 3), (1, 2, 2, 3)}
```

The [sum_of_squares\(\)](#) (page 628) routine will providean iterator that returns solutions and one may control whether the solutions contain zeros or not (and the solutions not containing zeros are returned first):

```
>>> from diofant.solvers.diophantine import sum_of_squares
>>> sos = sum_of_squares(18, 4, zeros=True)
```

(continues on next page)

(continued from previous page)

```
>>> next(sos)
(1, 2, 2, 3)
>>> next(sos)
(0, 0, 3, 3)
```

Simple Egyptian fractions can be found with the Diophantine module, too. For example, here are the ways that one might represent $1/2$ as a sum of two unit fractions:

```
>>> diophantine(Eq(1/x + 1/y, Rational(1, 2)))
{(-2, 1), (1, -2), (3, 6), (4, 4), (6, 3)}
```

To get a more thorough understanding of the Diophantine module, please refer to the following blog.

<https://thilinaatsympy.wordpress.com/>

References

- Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations
- Diophantine Equation, Wolfram Mathworld, [online]. Available: <https://mathworld.wolfram.com/DiophantineEquation.html>
- Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <https://web.archive.org/web/20181231080858/https://www.alpertron.com.ar/METHODS.HTM>
- Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <https://web.archive.org/web/20180831180321/http://www.jpr2718.org/ax2p.pdf>

User Functions

This function is imported into the global namespace with `from diofant import *`:

diophantine

```
diofant.solvers.diophantine.diophantine(eq, param=Symbol('t', integer=True),
                                           syms=None)
```

Simplify the solution procedure of diophantine equation *eq* by converting it into a product of terms which should equal zero.

$(x + y)(x - y) = 0$ and $x + y = 0$ and $x - y = 0$ are solved independently and combined. Each term is solved by calling `diop_solve()`.

Output of `diophantine()` is a set of tuples. The elements of the tuple are the solutions for each variable in the equation and are arranged according to the alphabetic ordering of the variables. e.g. For an equation with two variables, *a* and *b*, the first element of the tuple is the solution for *a* and the second for *b*.

Parameters

- **eq** (*Relational or Expr*) – an equation (to be solved)
- **t** (*Symbol, optional*) – the parameter to be used in the solution.

- **syms** (*list of Symbol's, optional*) - which determines the order of the elements in the returned tuple.

Examples

```
>>> diophantine(x**2 - y**2)
{(t_0, -t_0), (t_0, t_0)}
```

```
>>> diophantine(x*(2*x + 3*y - z))
{(0, n1, n2), (t_0, t_1, 2*t_0 + 3*t_1)}
>>> diophantine(x**2 + 3*x*y + 4*x)
{(0, n1), (3*t_0 - 4, -t_0)}
```

See also:

[*diofant.solvers.diophantine.diop_solve*](#) (page 617)

And this function is imported with `from diofant.solvers.diophantine import *`:

classify_diop

`diofant.solvers.diophantine.classify_diop(eq, _dict=True)`

Helper routine used by `diop_solve()` to find the type of the eq etc.

Parameters

eq (*Expr*) - an expression, which is assumed to be zero.

Examples

```
>>> classify_diop(4*x + 6*y - 4)
([x, y], {1: -4, x: 4, y: 6}, 'linear')
>>> classify_diop(x + 3*y - 4*z + 5)
([x, y, z], {1: 5, x: 1, y: 3, z: -4}, 'linear')
>>> classify_diop(x**2 + y**2 - x*y + x + 5)
([x, y], {1: 5, x: 1, x**2: 1, y**2: 1, x*y: -1}, 'binary_quadratic')
```

Returns

- Returns a tuple containing the type of the diophantine equation along with
- the variables (free symbols) and their coefficients. Variables are returned
- as a list and coefficients are returned as a dict with the key being the
- respective term and the constant term is keyed to `Integer(1)`. The type
- is one of the following -
 - `binary_quadratic`
 - `cubic_thue`
 - `general_pythagorean`
 - `general_sum_of_even_powers`
 - `general_sum_of_squares`

- homogeneous_general_quadratic
- homogeneous_ternary_quadratic
- homogeneous_ternary_quadratic_normal
- inhomogeneous_general_quadratic
- inhomogeneous_ternary_quadratic
- linear
- univariate

Internal Functions

These functions are intended for internal use in the Diophantine module.

diop_solve

`diofant.solvers.diophantine.diop_solve(eq, param=Symbol('t', integer=True))`

Solves the diophantine equation `eq`.

Unlike `diophantine()`, factoring of `eq` is not attempted. Uses `classify_diop()` to determine the type of the equation and calls the appropriate solver function.

Parameters

- **eq** (*Expr*) - an expression, which is assumed to be zero.
- **t** (*Symbol, optional*) - a parameter, to be used in the solution.

Examples

```
>>> from diofant.abc import w
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(t_0, 8*t_0 + 4*t_1 + 5, 7*t_0 + 3*t_1 + 5)
>>> diop_solve(x + 3*y - 4*z + w - 6)
(t_0, t_0 + t_1, 6*t_0 + 5*t_1 + 4*t_2 - 6, 5*t_0 + 4*t_1 + 3*t_2 - 6)
>>> diop_solve(x**2 + y**2 - 5)
{(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)}
```

See also:

[`diofant.solvers.diophantine.diophantine`](#) (page 615)

diop_linear

`diofant.solvers.diophantine.diop_linear(eq, param=Symbol('t', integer=True))`

Solves linear diophantine equations.

A linear diophantine equation is an equation of the form $a_1x_1 + a_2x_2 + \dots + a_nx_n = 0$ where a_1, a_2, \dots, a_n are integer constants and x_1, x_2, \dots, x_n are integer variables.

Parameters

- **eq** (*Expr*) - is a linear diophantine equation which is assumed to be zero.
- **param** (*Symbol, optional*) - is the parameter to be used in the solution.

Examples

```
>>> diop_linear(2*x - 3*y - 5)
(3*t_0 - 5, 2*t_0 - 5)
```

Here $x = -3*t_0 - 5$ and $y = -2*t_0 - 5$

```
>>> diop_linear(2*x - 3*y - 4*z - 3)
(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)
```

See also:

[`diofant.solvers.diophantine.diop_quadratic`](#) (page 619), [`diofant.solvers.diophantine.diop_ternary_quadratic`](#) (page 623), [`diofant.solvers.diophantine.diop_general_pythagorean`](#) (page 624), [`diofant.solvers.diophantine.diop_general_sum_of_squares`](#) (page 625)

base_solution_linear

`diofant.solvers.diophantine.base_solution_linear(c, a, b, t=None)`

Return the base solution for the linear equation, $ax + by = c$.

Used by `diop_linear()` to find the base solution of a linear Diophantine equation. If t is given then the parametrized solution is returned.

`base_solution_linear(c, a, b, t)`: a, b, c are coefficients in $ax + by = c$ and t is the parameter to be used in the solution.

Examples

```
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, -2*t + 5)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)
```

diop_quadratic

`diofant.solvers.diophantine.diop_quadratic(eq, param=Symbol('t', integer=True))`

Solves quadratic diophantine equations.

i.e. equations of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$. Returns a set containing the tuples (x, y) which contains the solutions. If there are no solutions then $(None, None)$ is returned.

Parameters

- **eq** (*Expr*) – should be a quadratic bivariate expression which is assumed to be zero.
- **param** (*Symbol, optional*) – is a parameter to be used in the solution.

Examples

```
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
{(-1, -1)}
```

References

- Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <https://web.archive.org/web/20181231080858/https://www.alpertron.com.ar/METHODS.HTM>
- Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <https://web.archive.org/web/20180831180321/http://www.jpr2718.org/ax2p.pdf>

See also:

`diofant.solvers.diophantine.diop_linear` (page 618), `diofant.solvers.diophantine.diop_ternary_quadratic` (page 623), `diofant.solvers.diophantine.diop_general_sum_of_squares` (page 625), `diofant.solvers.diophantine.diop_general_pythagorean` (page 624)

diop_DN

`diofant.solvers.diophantine.diop_DN(D, N, t=Symbol('t', integer=True))`

Solves the equation $x^2 - Dy^2 = N$.

Mainly concerned with the case $D > 0$, D is not a perfect square, which is the same as the generalized Pell equation. The LMM algorithm is used to solve this equation.

Returns

- A tuple of pairs, (x, y) , for each class of the solutions.
- *Other solutions of the class can be constructed according to the*
- values of D and N .

Parameters

- **D, N** (*Integer*) – correspond to D and N in the equation.

- **t** (*Symbol, optional*) - is the parameter to be used in the solutions.

Examples

```
>>> diop_DN(13, -4) # Solves equation x**2 - 13*y**2 = -4
[(3, 1), (393, 109), (36, 10)]
```

The output can be interpreted as follows: There are three fundamental solutions to the equation $x^2 - 13y^2 = -4$ given by (3, 1), (393, 109) and (36, 10). Each tuple is in the form (x, y), i.e. solution (3, 1) means that $x = 3$ and $y = 1$.

```
>>> diop_DN(986, 1) # Solves equation x**2 - 986*y**2 = 1
[(49299, 1570)]
```

See also:

`diofant.solvers.diophantine.find_DN` (page 623), `diofant.solvers.diophantine.diop_bf_DN` (page 621)

References

- Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Pages 16 - 17. [online], Available: <https://web.archive.org/web/20180831180333/http://www.jpr2718.org/pell.pdf>

cornacchia

`diofant.solvers.diophantine.cornacchia(a, b, m)`

Solves $ax^2 + by^2 = m$ where $\gcd(a, b) = 1 = \gcd(a, m)$ and $a, b > 0$.

Uses the algorithm due to Cornacchia. The method only finds primitive solutions, i.e. ones with $\gcd(x, y) = 1$. So this method can't be used to find the solutions of $x^2 + y^2 = 20$ since the only solution to former is $(x, y) = (4, 2)$ and it is not primitive. When $a = b$, only the solutions with $x \leq y$ are found. For more details, see the References.

Examples

```
>>> cornacchia(2, 3, 35)
{(2, 3), (4, 1)}
>>> cornacchia(1, 1, 25)
{(4, 3)}
```


References

- A. Nitaj, “L’algorithme de Cornacchia”
- Solving the diophantine equation $ax^2 + by^2 = m$ by Cornacchia’s method, [online], Available: <http://www.numbertheory.org/php/cornacchia.html>

See also:

`diofant.utilities.iterables.signed_permutations` (page 743)

diop_bf_DN

`diofant.solvers.diophantine.diop_bf_DN(D, N, t=Symbol('t', integer=True))`

Uses brute force to solve the equation, $x^2 - Dy^2 = N$.

Mainly concerned with the generalized Pell equation which is the case when $D > 0$, D is not a perfect square. Let (t, u) be the minimal positive solution of the equation $x^2 - Dy^2 = 1$.

Then this method requires $\sqrt{\frac{|N|(t \pm 1)}{2D}}$ to be small.

Parameters

- **D, N** (*Integer*) – correspond to D and N in the equation.
- **t** (*Symbol, optional*) – is the parameter to be used in the solutions.

Examples

```
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

See also:

`diofant.solvers.diophantine.diop_DN` (page 619)

References

- Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Page 15. <https://web.archive.org/web/20180831180333/http://www.jpr2718.org/pell.pdf>

transformation_to_DN

`diofant.solvers.diophantine.transformation_to_DN(eq)`

This function transforms general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$ to more easy to deal with $X^2 - DY^2 = N$ form.

This is used to solve the general quadratic equation by transforming it to the latter form. This function returns a tuple (A, B) where A is a 2 X 2 matrix and B is a 2 X 1 matrix such that,

$$\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$$

Parameters

eq (*Expr*) – the quadratic expression to be transformed.

Examples

```
>>> A, B = transformation_to_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
>>> A
Matrix([
[1/26, 3/26]
[ 0, 1/13]])
>>> B
Matrix([
[-6/13]
[-4/13]])
```

A, B returned are such that $\text{Transpose}((x\ y)) = A * \text{Transpose}((X\ Y)) + B$. Substituting these values for x and y and a bit of simplifying work will give an equation of the form $x^2 - Dy^2 = N$.

```
>>> from diofant.abc import X, Y
>>> u = (A*Matrix([X, Y]) + B)[0] # Transformation for x
>>> u
X/26 + 3*Y/26 - 6/13
>>> v = (A*Matrix([X, Y]) + B)[1] # Transformation for y
>>> v
Y/13 - 4/13
```

Next we will substitute these formulas for x and y and do `simplify()`.

```
>>> eq = simplify((x**2 - 3*x*y - y**2 - 2*y + 1).subs({x: u, y: v}))
>>> eq
X**2/676 - Y**2/52 + 17/13
```

By multiplying the denominator appropriately, we can get a Pell equation in the standard form.

```
>>> eq * 676
X**2 - 13*Y**2 + 884
```

If only the final equation is needed, `find_DN()` can be used.

See also:

`diofant.solvers.diophantine.find_DN` (page 623)

References

- Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, John P. Robertson, May 8, 2003, Page 7 - 11. <https://web.archive.org/web/20180831180321/http://www.jpr2718.org/ax2p.pdf>

find_DN

`diofant.solvers.diophantine.find_DN(eq)`

This function returns a tuple, (D, N) of the simplified form, $x^2 - Dy^2 = N$, corresponding to the general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

Solving the general quadratic is then equivalent to solving the equation $X^2 - DY^2 = N$ and transforming the solutions by using the transformation matrices returned by `transformation_to_DN()`.

Parameters

eq (*Expr*) - is the quadratic expression to be transformed.

Examples

```
>>> find_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
(13, -884)
```

Interpretation of the output is that we get $X^2 - 13Y^2 = -884$ after transforming $x^2 - 3xy - y^2 - 2y + 1$ using the transformation returned by `transformation_to_DN()`.

See also:

`diofant.solvers.diophantine.transformation_to_DN` (page 621)

References

- Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, John P.Robertson, May 8, 2003, Page 7 - 11. <https://web.archive.org/web/20180831180321/http://www.jpr2718.org/ax2p.pdf>

diop_ternary_quadratic

`diofant.solvers.diophantine.diop_ternary_quadratic(eq)`

Solves the general quadratic ternary form, $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

Returns

tuple - which is a base solution for the above equation. If there are no solutions, `(None, None, None)` is returned.

Parameters

eq (*Expr*) - should be an homogeneous expression of degree two in three variables and it is assumed to be zero.

Examples

```
>>> diop_ternary_quadratic(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic(45*x**2 - 7*y**2 - 8*x*y - z**2)
(28, 45, 105)
>>> diop_ternary_quadratic(x**2 - 49*y**2 - z**2 + 13*z*y - 8*x*y)
(9, 1, 5)
```

descent

`diofant.solvers.diophantine.descent(A, B)`

Returns a non-trivial solution, (x, y, z) , to $x^2 = Ay^2 + Bz^2$ using Lagrange's descent method with lattice-reduction. A and B are assumed to be valid for such a solution to exist.

This is faster than the normal Lagrange's descent algorithm because the Gaussian reduction is used.

Examples

```
>>> descent(3, 1) # x**2 = 3*y**2 + z**2
(1, 0, 1)
```

$(x, y, z) = (1, 0, 1)$ is a solution to the above equation.

```
>>> descent(41, -113)
(-16, -3, 1)
```

References

- Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.

diop_general_pythagorean

`diofant.solvers.diophantine.diop_general_pythagorean(eq, param=Symbol('m', integer=True))`

Solves the general pythagorean equation, $a_1^2x_1^2 + a_2^2x_2^2 + \dots + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0$.

Returns a tuple which contains a parametrized solution to the equation, sorted in the same order as the input variables.

Parameters

- **eq** (*Expr*) - is a general pythagorean equation which is assumed to be zero
- **param** (*Symbol, optional*) - is the base parameter used to construct other parameters by subscripting.

Examples

```
>>> from diofant.abc import e
>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2 + 10*m2**2 + 10*m3**2 - 10*m4**2, 15*m1**2 + 15*m2**2 + 15*m3**2 +
↪ 15*m4**2, 15*m1*m4, 12*m2*m4, 60*m3*m4)
```

diop_general_sum_of_squares

`diofant.solvers.diophantine.diop_general_sum_of_squares(eq, limit=1)`

Solves the equation $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$.

Returns at most limit number of solutions.

Parameters

- **eq** (*Expr*) - is an expression which is assumed to be zero. Also, eq should be in the form, $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$.
- **limit** (*int, optional*) - upper limit (the default is 1) for number of solutions returned.

Notes

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$ then there will be no solutions.

Examples

```
>>> from diofant.abc import e
>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345)
{(15, 22, 22, 24, 24)}
```

References

- Representing an integer as a sum of three squares, [online], Available: https://proofwiki.org/wiki/Integer_as_Sum_of_Three_Squares

diop_general_sum_of_even_powers

`diofant.solvers.diophantine.diop_general_sum_of_even_powers(eq, limit=1)`

Solves the equation $x_1^e + x_2^e + \dots + x_n^e - k = 0$ where e is an even, integer power.

Returns at most limit number of solutions.

Parameters

- **eq** (*Expr*) - An expression which is assumed to be zero. Also, eq should be in the form, $x_1^e + x_2^e + \dots + x_n^e - k = 0$.
- **limit** (*Expr, optional*) - Limit number of returned solutions. Default is 1.

Examples

```
>>> diop_general_sum_of_even_powers(a**4 + b**4 - (2**4 + 3**4))
{(2, 3)}
```

See also:

[*diofant.solvers.diophantine.power_representation*](#) (page 628)

partition

`diofant.solvers.diophantine.partition(n, k=None, zeros=False)`

Returns a generator that can be used to generate partitions of an integer *n*.

A partition of *n* is a set of positive integers which add up to *n*. For example, partitions of 3 are 3, 1 + 2, 1 + 1 + 1. A partition is returned as a tuple. If *k* equals None, then all possible partitions are returned irrespective of their size, otherwise only the partitions of size *k* are returned. If the *zero* parameter is set to True then a suitable number of zeros are added at the end of every partition of size less than *k*.

Parameters

- **n** (*int*) – is a positive integer
- **k** (*int, optional*) – is the size of the partition which is also positive integer. The default is None.
- **zeros** (*boolean, optional*) – parameter is considered only if *k* is not None. When the partitions are over, the last *next()* call throws the `StopIteration` exception, so this function should always be used inside a `try` - `except` block.

Examples

```
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(1, 1, 3)
>>> next(g)
(1, 2, 2)
>>> g = partition(5, 3, zeros=True)
>>> next(g)
(0, 0, 5)
```

sum_of_three_squares

`diofant.solvers.diophantine.sum_of_three_squares(n)`

Returns a 3-tuple (a, b, c) such that $a^2 + b^2 + c^2 = n$ and $a, b, c \geq 0$.

Returns None if $n = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$.

Parameters

n (*int*) – a non-negative integer.

Examples

```
>>> sum_of_three_squares(44542)
(18, 37, 207)
```

References

- Representing a number as a sum of three squares, [online], Available: <https://schorn.ch/lagrange.html>

See also:

`diofant.solvers.diophantine.sum_of_squares` (page 628)

sum_of_four_squares

`diofant.solvers.diophantine.sum_of_four_squares(n)`

Returns a 4-tuple (a, b, c, d) such that $a^2 + b^2 + c^2 + d^2 = n$.

Here $a, b, c, d \geq 0$.

Parameters

n (*int*) – is a non-negative integer.

Examples

```
>>> sum_of_four_squares(3456)
(8, 8, 32, 48)
>>> sum_of_four_squares(1294585930293)
(0, 1234, 2161, 1137796)
```

References

- Representing a number as a sum of four squares, [online], Available: <https://schorn.ch/lagrange.html>

See also:

`diofant.solvers.diophantine.sum_of_squares` (page 628)

power_representation

`diofant.solvers.diophantine.power_representation(n, p, k, zeros=False)`

Returns a generator for finding k-tuples of integers, (n_1, n_2, \dots, n_k) , such that $n = n_1^p + n_2^p + \dots + n_k^p$.

Parameters

- **n** (*int*) – a non-negative integer
- **k, p** (*int*) – parameters to control representation n as a sum of k, p-th powers.
- **zeros** (*boolean, optional*) – if True (the default is False), then the solutions will contain zeros.

Examples

Represent 1729 as a sum of two cubes:

```
>>> f = power_representation(1729, 3, 2)
>>> next(f)
(9, 10)
>>> next(f)
(1, 12)
```

If the flag `zeros` is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even p the `permute_sign` function can be used to get all signed values:

```
>>> from diofant.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```
>>> from diofant.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1),
 (12, -1), (-12, -1)]
```

`diofant.solvers.diophantine.sum_of_powers()`

alias of `power_representation()` (page 628)

sum_of_squares

`diofant.solvers.diophantine.sum_of_squares(n, k, zeros=False)`

Return a generator that yields the k-tuples of nonnegative values, the squares of which sum to n. If `zeros` is False (default) then the solution will not contain zeros. The nonnegative elements of a tuple are sorted.

- If $k == 1$ and n is square, $(n,)$ is returned.

- If $k == 2$ then n can only be written as a sum of squares if every prime in the factorization of n that has the form $4*k + 3$ has an even multiplicity. If n is prime then it can only be written as a sum of two squares if it is in the form $4*k + 1$.
- if $k == 3$ then n can be written as a sum of squares if it does not have the form $4*m*(8*k + 7)$.
- all integers can be written as the sum of 4 squares.
- if $k > 4$ then n can be partitioned and each partition can be written as a sum of 4 squares; if n is not evenly divisible by 4 then n can be written as a sum of squares only if the an additional partition can be written as as sum of squares. For example, if $k = 6$ then n is partitioned into two parts, the first being written as a sum of 4 squares and the second being written as a sum of 2 squares - which can only be done if the contition above for $k = 2$ can be met, so this will automatically reject certain partitions of n .

Examples

```
>>> list(sum_of_squares(25, 2))
[(3, 4)]
>>> list(sum_of_squares(25, 2, True))
[(3, 4), (0, 5)]
>>> list(sum_of_squares(25, 4))
[(1, 2, 2, 4)]
```

See also:

[*diofant.utilities.iterables.signed_permutations*](#) (page 743)

merge_solution

`diofant.solvers.diophantine.merge_solution(var, var_t, solution)`

This is used to construct the full solution from the solutions of sub equations.

For example when solving the equation $(x - y)(x^2 + y^2 - z^2) = 0$, solutions for each of the equations $x - y = 0$ and $x^2 + y^2 - z^2$ are found independently. Solutions for $x - y = 0$ are $(x, y) = (t, t)$. But we should introduce a value for z when we output the solution for the original equation. This function converts (t, t) into (t, t, n_1) where n_1 is an integer parameter.

divisible

`diofant.solvers.diophantine.divisible(a, b)`

Returns *True* if a is divisible by b and *False* otherwise.

PQa

`diofant.solvers.diophantine.PQa(P_0, Q_0, D)`

Returns useful information needed to solve the Pell equation.

There are six sequences of integers defined related to the continued fraction representation of $\frac{P+\sqrt{D}}{Q}$, namely $\{P_i\}$, $\{Q_i\}$, $\{a_i\}$, $\{A_i\}$, $\{B_i\}$, $\{G_i\}$. `PQa()` Returns these values as a 6-tuple in the same order as mentioned above.

Parameters

P_0, Q_0, D (*Integer*) - integers corresponding to P_0 , Q_0 and D in the continued fraction $\frac{P_0+\sqrt{D}}{Q_0}$. Also it's assumed that $P_0^2 \equiv D \pmod{|Q_0|}$ and D is square free.

Examples

```
>>> pqa = PQa(13, 4, 5) # (13 + sqrt(5))/4
>>> next(pqa) # (P_0, Q_0, a_0, A_0, B_0, G_0)
(13, 4, 3, 3, 1, -1)
>>> next(pqa) # (P_1, Q_1, a_1, A_1, B_1, G_1)
(-1, 1, 1, 4, 1, 3)
```

References

- Solving the generalized Pell equation $x^2 - Dy^2 = N$, John P. Robertson, July 31, 2004, Pages 4 - 8. <https://web.archive.org/web/20180831180333/http://www.jpr2718.org/pell.pdf>

equivalent

`diofant.solvers.diophantine.equivalent(u, v, r, s, D, N)`

Returns True if two solutions (u, v) and (r, s) of $x^2 - Dy^2 = N$ belongs to the same equivalence class and False otherwise.

Two solutions (u, v) and (r, s) to the above equation fall to the same equivalence class iff both $(ur - Dvs)$ and $(us - vr)$ are divisible by N . No test is performed to test whether (u, v) and (r, s) are actually solutions to the equation. User should take care of this.

Parameters

u, v, r, s, D, N (*Integer*)

Examples

```
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

References

- Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Page 12. <https://web.archive.org/web/20180831180333/http://www.jpr2718.org/pell.pdf>

parametrize_ternary_quadratic

`diofant.solvers.diophantine.parametrize_ternary_quadratic(eq)`

Returns the parametrized general solution for the ternary quadratic equation `eq` which has the form $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

Examples

```
>>> parametrize_ternary_quadratic(x**2 + y**2 - z**2)
(2*p*q, p**2 - q**2, p**2 + q**2)
```

Here p and q are two co-prime integers.

```
>>> parametrize_ternary_quadratic(3*x**2 + 2*y**2 - z**2 - 2*x*y + 5*y*z - 7*y*z)
(2*p**2 - 2*p*q - q**2, 2*p**2 + 2*p*q - q**2, 2*p**2 - 2*p*q + 3*q**2)
>>> parametrize_ternary_quadratic(124*x**2 - 30*y**2 - 7729*z**2)
(-1410*p**2 - 363263*q**2, 2700*p**2 + 30916*p*q - 695610*q**2, -60*p**2 +
↪ 5400*p*q + 15458*q**2)
```

References

- The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.

diop_ternary_quadratic_normal

`diofant.solvers.diophantine.diop_ternary_quadratic_normal(eq)`

Solves the quadratic ternary diophantine equation, $ax^2 + by^2 + cz^2 = 0$.

Here the coefficients a , b , and c should be non zero. Otherwise the equation will be a quadratic binary or univariate equation. If solvable, returns a tuple (x, y, z) that satisfies the given equation. If the equation does not have integer solutions, $(None, None, None)$ is returned.

Examples

```
>>> diop_ternary_quadratic_normal(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic_normal(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic_normal(34*x**2 - 3*y**2 - 301*z**2)
(4, 9, 1)
```

ldescent

`diofant.solvers.diophantine.ldescent(A, B)`

Return a non-trivial solution to $w^2 = Ax^2 + By^2$ using Lagrange's method; return None if there is no such solution.

Here, $A \neq 0$ and $B \neq 0$ and A and B are square free. Output a tuple (w_0, x_0, y_0) which is a solution to the above equation.

Examples

```
>>> ldescent(1, 1) # w^2 = x^2 + y^2
(1, 1, 0)
>>> ldescent(4, -7) # w^2 = 4x^2 - 7y^2
(2, -1, 0)
```

This means that $x = -1, y = 0$ and $w = 2$ is a solution to the equation $w^2 = 4x^2 - 7y^2$

```
>>> ldescent(5, -1) # w^2 = 5x^2 - y^2
(2, 1, -1)
```

References

- The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0, <http://eprints.nottingham.ac.uk/60/1/kvxefz87.pdf>

gaussian_reduce

`diofant.solvers.diophantine.gaussian_reduce(w, a, b)`

Returns a reduced solution (x, z) to the congruence $X^2 - aZ^2 \equiv 0 \pmod{b}$ so that $x^2 + |a|z^2$ is minimal.

Here w is a solution of the congruence $x^2 \equiv a \pmod{b}$

References

- Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.

holzer

`diofant.solvers.diophantine.holzer(x, y, z, a, b, c)`

Simplify the solution (x, y, z) of the equation $ax^2 + by^2 = cz^2$ with $a, b, c > 0$ and $z^2 \geq |ab|$ to a new reduced solution (x', y', z') such that $z'^2 \leq |ab|$.

The algorithm is an interpretation of Mordell's reduction as described on page 8 of Cremona and Rusin's paper and the work of Mordell.

References

- Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- Diophantine Equations, L. J. Mordell, page 48.

prime_as_sum_of_two_squares

`diofant.solvers.diophantine.prime_as_sum_of_two_squares(p)`

Represent a prime p as a unique sum of two squares; this can only be done if the prime is congruent to 1 mod 4.

Examples

```
>>> prime_as_sum_of_two_squares(7) # can't be done
>>> prime_as_sum_of_two_squares(5)
(1, 2)
```

References

- Representing a number as a sum of four squares, [online], Available: <https://schorn.ch/lagrange.html>

See also:

`diofant.solvers.diophantine.sum_of_squares` (page 628)

square_factor

`diofant.solvers.diophantine.square_factor(a)`

Returns an integer c s.t. $a = c^2k$, $c, k \in \mathbb{Z}$. Here k is square free. a can be given as an integer or a dictionary of factors.

Examples

```
>>> square_factor(24)
2
>>> square_factor(-36*3)
6
>>> square_factor(1)
1
>>> square_factor({3: 2, 2: 1, -1: 1})
3
```

See also:

diofant.solvers.diophantine.reconstruct (page 634), *diofant.ntheory.factor_core* (page 236)

sqf_normal

`diofant.solvers.diophantine.sqf_normal(a, b, c, steps=False)`

Return a', b', c' , the coefficients of the square-free normal form of $ax^2 + by^2 + cz^2 = 0$, where a', b', c' are pairwise prime. If *steps* is True then also return three tuples: *sq*, *sqf*, and (a', b', c') where *sq* contains the square factors of a , b and c after removing the $\gcd(a, b, c)$; *sqf* contains the values of a , b and c after removing both the $\gcd(a, b, c)$ and the square factors.

The solutions for $ax^2 + by^2 + cz^2 = 0$ can be recovered from the solutions of $a'x^2 + b'y^2 + c'z^2 = 0$.

Examples

```
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11)
(11, 1, 5)
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11, True)
((3, 1, 7), (5, 55, 11), (11, 1, 5))
```

References

- Legendre's Theorem, Lefrange's Descent, https://public.csusm.edu/aitken_html/notes/legendre.pdf

See also:

diofant.solvers.diophantine.reconstruct (page 634)

reconstruct

`diofant.solvers.diophantine.reconstruct(A, B, z)`

Reconstruct the z value of an equivalent solution of $ax^2 + by^2 + cz^2$ from the z value of a solution of the square-free normal form of the equation, $a' * x^2 + b' * y^2 + c' * z^2$, where a' , b' and c' are square free and $\gcd(a', b', c') == 1$.

transformation_to_normal

`diofant.solvers.diophantine.transformation_to_normal(eq)`

Returns the transformation Matrix that converts a general ternary quadratic equation eq ($ax^2 + by^2 + cz^2 + dxy + eyz + fzx$) to a form without cross terms: $ax^2 + by^2 + cz^2 = 0$. This is not used in solving ternary quadratics; it is only implemented for the sake of completeness.

4.16.4 ODE

User Functions

These are functions that are imported into the global namespace with `from diofant import *`. These functions (unlike [Hint Functions](#) (page 643), below) are intended for use by ordinary users of Diofant.

dsolve

`diofant.solvers.ode.dsolve(eq, func=None, hint='default', simplify=True, init=None, xi=None, eta=None, x0=0, n=6, **kwargs)`

Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.

For single ordinary differential equation

It is classified under this when number of equation in `eq` is one.

Usage

`dsolve(eq, f(x), hint)` -> Solve ordinary differential equation `eq` for function `f(x)`, using method `hint`.

Details

eq can be any supported ordinary differential equation (see the [ode](#) (page 680) docstring for supported methods). This can either be an [Equality](#) (page 109), 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 638). See Hints below for more options that you can use for hint.

simplify enables simplification by `odesimp()` (page 643). See its docstring for more information. Turn this off, for example, to disable solving of solutions for `func` or simplification of arbitrary constants. It will still integrate with this hint. Note that the solution may contain more arbitrary constants than the order of the ODE with this option enabled.

xi and eta are the infinitesimal functions of an ordinary

differential equation. They are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. The user can specify values for the infinitesimals. If nothing is specified, xi and eta are calculated using [infinitesimals\(\)](#) (page 641) with the help of various heuristics.

init is the set of initial/boundary conditions for the differential equation.

It should be given in the form of `{f(x0): x1, f(x).diff(x).subs({x: x2}): x3}` and so on. For power series solutions, if no initial conditions are specified `f(0)` is assumed to be `C0` and the power series solution is calculated about 0.

x0 is the point about which the power series solution of a differential equation is to be evaluated.

n gives the exponent of the dependent variable up to which the power series

solution of a differential equation is to be evaluated.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to [dsolve\(\)](#) (page 635):

default:

This uses whatever hint is returned first by [classify_ode\(\)](#) (page 638). This is the default argument to [dsolve\(\)](#) (page 635).

all:

To make [dsolve\(\)](#) (page 635) 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 696) 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 638) 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 638).

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 635) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than `all`, because [integrate\(\)](#) (page 69) is an expensive routine.

best:

To have [dsolve\(\)](#) (page 635) try all methods and return the simplest one. This takes into account whether the solution is solvable in the function,

whether it contains any Integral classes (i.e. unevaluable integrals), and which one is the shortest in size.

See also the `classify_ode()` (page 638) docstring for more info on hints, and the `ode` (page 680) docstring for a list of all supported hints.

Tips

- See `test_ode.py` for many tests, which serves also as a set of examples for how to use `dsolve()` (page 635).
- `dsolve()` (page 635) always returns an `Equality` (page 109) class (except for the case when the hint is `all` or `all_Integral`). If possible, it solves the solution explicitly for the function being solved for. Otherwise, it returns an implicit solution.
- Arbitrary constants are symbols named `C1`, `C2`, and so on.
- Because all solutions should be mathematically equivalent, some hints may return the exact same result for an ODE. Often, though, two different hints will return the same solution formatted differently. The two should be equivalent. Also note that sometimes the values of the arbitrary constants in two different solutions may not be the same, because one constant may have “absorbed” other constants into it.
- Do `help(ode.ode_<hintname>)` to get help more information on a specific hint, where `<hintname>` is the name of a hint without `_Integral`.

For system of ordinary differential equations

Usage

`dsolve(eq, func)` -> Solve a system of ordinary differential equations `eq` for `func` being list of functions including $x(t)$, $y(t)$, $z(t)$ where number of functions in the list depends upon the number of equations provided in `eq`.

Details

`eq` can be any supported system of ordinary differential equations This can either be an `Equality` (page 109), or an expression, which is assumed to be equal to 0.

`func` holds $x(t)$ and $y(t)$ being functions of one variable which together with some of their derivatives make up the system of ordinary differential equation `eq`. It is not necessary to provide this; it will be autodetected (and an error raised if it couldn't be detected).

Hints

The hints are formed by parameters returned by `classify_sysode`, combining them give hints name used later for forming method name.

Examples

```
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))
```

```
>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='lst_exact')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> dsolve(eq, hint='almost_linear')
[Eq(f(x), -acos(C1/sqrt(-cos(x)**2)) + 2*pi), Eq(f(x), acos(C1/sqrt(-cos(x)**2)))]
>>> eq = (Eq(Derivative(f(t), t), 12*t*f(t) + 8*g(t)),
```

(continues on next page)

(continued from previous page)

```

...      Eq(Derivative(g(t), t), 21*f(t) + 7*t*g(t))
>>> dsolve(eq)
[Eq(f(t), C1*x0(t) + C2*x0(t)*Integral(8*E**Integral(7*t, t)*E**Integral(12*t, t)/
->x0(t)**2, t)),
Eq(g(t), C1*y0(t) + C2*(E**Integral(7*t, t)*E**Integral(12*t, t)/x0(t) +
y0(t)*Integral(8*E**Integral(7*t, t)*E**Integral(12*t, t)/x0(t)**2, t)))]
>>> eq = (Eq(Derivative(f(t), t), f(t)*g(t)*sin(t)),
...      Eq(Derivative(g(t), t), g(t)**2*sin(t)))
>>> dsolve(eq)
{Eq(f(t), -E**C1/(E**C1*C2 - cos(t))), Eq(g(t), -1/(C1 - cos(t)))}

```

classify_ode

`diofant.solvers.ode.classify_ode(eq, func=None, dict=False, init=None, **kwargs)`

Returns a tuple of possible `dsolve()` (page 635) classifications for an ODE.

The tuple is ordered so that first item is the classification that `dsolve()` (page 635) uses to solve the ODE by default. In general, classifications at the near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make `dsolve()` (page 635) use a different classification, use `dsolve(ODE, func, hint=<classification>)`. See also the `dsolve()` (page 635) docstring for different meta-hints you can use.

If `dict` is true, `classify_ode()` (page 638) will return a dictionary of `hint:match` expression terms. This is intended for internal use by `dsolve()` (page 635). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hint-name` is the name of the hint without `_Integral`.

See [allhints](#) (page 643) or the [ode](#) (page 680) docstring for a list of all supported hints that can be returned from `classify_ode()` (page 638).

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 405) class in it. Note that a hint may do this anyway if `integrate()` (page 399) 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 399) is an expensive routine. If `dsolve()` (page 635) hangs, it is probably because `integrate()` (page 69) 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()` (page 399) 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 405)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

```
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('separable', '1st_linear', '1st_homogeneous_coeff_best',
 '1st_homogeneous_coeff_subs_indep_div_dep',
 '1st_homogeneous_coeff_subs_dep_div_indep',
 '1st_power_series', 'lie_group',
 'nth_linear_constant_coeff_homogeneous',
 'separable_Integral', '1st_linear_Integral',
 '1st_homogeneous_coeff_subs_indep_div_dep_Integral',
 '1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> Classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('nth_linear_constant_coeff_undetermined_coefficients',
 'nth_linear_constant_coeff_variation_of_parameters',
 'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

checkodesol

`diofant.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 109) that the solution satisfies, e.g. `Eq(f(x), C1)`, `Eq(f(x) + C1, 0)`; or simply an *Expr* (page 57), e.g. `f(x) - C1`. In most cases it will not be necessary to explicitly identify the function, but if the function cannot be inferred from the original equation it can be supplied through the `func` argument.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

It tries the following methods, in order, until it finds zero equivalence:

1. Substitute the solution for f in the original equation. This only works if `ode` is solved for f . It will attempt to solve it first unless `solve_for_func == False`.
2. Take n derivatives of the solution, where n is the order of `ode`, and check to see if that is equal to the solution. This only works on exact ODEs.
3. Take the 1st, 2nd, ..., n th derivatives of the solution, each time solving for the derivative of f of that order (this will always be possible because f is a linear operator). Then back substitute each derivative into `ode` in reverse order.

This function returns a tuple. The first item in the tuple is `True` if the substitution results in 0, and `False` otherwise. The second item in the tuple is what the substitution results in. It should always be 0 if the first item is `True`. 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 580) 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

```
>>> C1 = symbols('C1')
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> checkodesol(f(x).diff(x), C1)
(True, 0)
>>> checkodesol(f(x).diff(x), x)
(False, 1)
>>> checkodesol(f(x).diff((x, 2)), x**2)
(False, 2)
```

homogeneous_order

`diofant.solvers.ode.homogeneous_order(eq, *symbols)`

Returns the order n if g is homogeneous and `None` if it is not homogeneous.

Determines if a function is homogeneous and if so of what order. A function $f(x, y, \dots)$ is homogeneous of order n if $f(tx, ty, \dots) = t^n f(x, y, \dots)$.

If the function is of two variables, $F(x, y)$, then f being homogeneous of any order is equivalent to being able to rewrite $F(x, y)$ as $G(x/y)$ or $H(y/x)$. This fact is used to solve 1st order ordinary differential equations whose coefficients are homogeneous of the same order (see the docstrings of [ode_1st_homogeneous_coeff_subs_dep_div_indep\(\)](#) (page 649) and [ode_1st_homogeneous_coeff_subs_indep_div_dep\(\)](#) (page 650)).

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

```
>>> 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

`diofant.solvers.ode.infinitesimals(eq, func=None, order=None, hint='default', match=None)`

The infinitesimal functions of an ordinary differential equation, $\xi(x, y)$ and $\eta(x, y)$, are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. So, the ODE $y' = f(x, y)$ would admit a Lie group $x^* = X(x, y; \varepsilon) = x + \varepsilon \xi(x, y)$, $y^* = Y(x, y; \varepsilon) = y + \varepsilon \eta(x, y)$ such that $(y^*)' = f(x^*, y^*)$. A change of coordinates, to $r(x, y)$ and $s(x, y)$, can be performed so this Lie group becomes the translation group, $r^* = r$ and $s^* = s + \varepsilon$. They are tangents to the coordinate curves of the new system.

Consider the transformation $(x, y) \rightarrow (X, Y)$ such that the differential equation remains invariant. ξ and η are the tangents to the transformed coordinates X and Y , at $\varepsilon = 0$.

$$\left(\frac{\partial X(x, y; \varepsilon)}{\partial \varepsilon} \right) \Big|_{\varepsilon=0} = \xi, \quad \left(\frac{\partial Y(x, y; \varepsilon)}{\partial \varepsilon} \right) \Big|_{\varepsilon=0} = \eta,$$

The infinitesimals can be found by solving the following PDE:

```
>>> xi, eta = map(Function, ['xi', 'eta'])
>>> h = h(x, y) # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq(eta.diff(x) + (eta.diff(y) - xi.diff(x))*h -
...              (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
```

(continues on next page)

(continued from previous page)

```
>>> pprint(genform)
- \eta(x, y) \cdot \frac{\partial}{\partial y}(h(x, y)) - h^2(x, y) \cdot \frac{\partial}{\partial y}(\xi(x, y)) + h(x, y) \cdot \left( \frac{\partial}{\partial y}(\eta(x, y)) - \frac{\partial}{\partial x}(\xi(x, y)) \right) - \xi(x, y) \cdot \frac{\partial}{\partial x}(h(x, y)) + \frac{\partial}{\partial x}(\eta(x, y)) = 0
```

Solving the above mentioned PDE is not trivial, and can be solved only by making intelligent assumptions for ξ and η (heuristics). Once an infinitesimal is found, the attempt to find more heuristics stops. This is done to optimise the speed of solving the differential equation. If a list of all the infinitesimals is needed, `hint` should be flagged as `all`, which gives the complete list of infinitesimals. If the infinitesimals for a particular heuristic needs to be found, it can be passed as a flag to `hint`.

Examples

```
>>> eta = Function('eta')
>>> xi = Function('xi')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq)
[{eta(x, f(x)): E**(x**3/3), xi(x, f(x)): 0}]
```

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

checkinfosol

`diofant.solvers.ode.checkinfosol(eq, infinitesimals, func=None, order=None)`

This function is used to check if the given infinitesimals are the actual infinitesimals of the given first order differential equation. This method is specific to the Lie Group Solver of ODEs.

As of now, it simply checks, by substituting the infinitesimals in the partial differential equation.

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi \frac{\partial h}{\partial x} - \eta \frac{\partial h}{\partial y} = 0$$

where η , and ξ are the infinitesimals and $h(x, y) = \frac{dy}{dx}$

The infinitesimals should be given in the form of a list of dicts `[{xi(x, y): inf, eta(x, y): inf}]`, corresponding to the output of the function `infinitesimals`. It returns a list of values of the form `[(True/False, sol)]` where `sol` is the value obtained after substituting the infinitesimals in the PDE. If it is `True`, then `sol` would be 0.

Hint Functions

These functions are intended for internal use by `dsolve()` (page 635) and others. Unlike *User Functions* (page 635), above, these are not intended for every-day use by ordinary Diofant users. Instead, functions such as `dsolve()` (page 635) 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

```
diofant.solvers.ode.allhints = ('separable', '1st_exact', '1st_linear',
'Bernoulli', 'Riccati_special_minus2', '1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep', 'almost_linear',
'linear_coefficients', 'separable_reduced', '1st_power_series', 'lie_group',
'nth_linear_constant_coeff_homogeneous', 'nth_linear_euler_eq_homogeneous',
'nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters', 'Liouville',
'2nd_power_series_ordinary', '2nd_power_series_regular', 'separable_Integral',
'1st_exact_Integral', '1st_linear_Integral', 'Bernoulli_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral', 'almost_linear_Integral',
'linear_coefficients_Integral', 'separable_reduced_Integral',
'nth_linear_constant_coeff_variation_of_parameters_Integral',
'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters_Integral',
'Liouville_Integral')
```

This is a list of hints in the order that they should be preferred by `classify_ode()` (page 638). 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 Diofant.

The hint used by `dsolve()` (page 635) 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

```
diofant.solvers.ode.odesimp(eq, func, order, constants, hint)
```

Simplifies ODEs, including trying to solve for `func` and running `constantsimp()` (page 645).

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 405)s in the expression, if the hint is not an `_Integral` hint.

This function should have no effect on expressions returned by `dsolve()` (page 635), as `dsolve()` (page 635) already calls `odesimp()` (page 643), but the individual hint functions do not call `odesimp()` (page 643) (because the `dsolve()` (page 635) wrapper does). Therefore, this function is designed for mainly internal use.

Examples

```
>>> C1 = symbols('C1')
```

```
>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x),
...             hint='lst_homogeneous_coeff_subs_indep_div_dep_Integral',
...             simplify=False)
>>> pprint(eq, wrap_line=False)
```

$$\log(f(x)) = \log(C_1) + \frac{\frac{x}{f(x)} - \left(u_2 + \frac{1}{\sin\left(\frac{1}{u_2}\right)} \right)}{u_2^2} d(u_2)$$

```
>>> pprint(odesimp(eq, f(x), 1, {C1},
...                 hint='lst_homogeneous_coeff_subs_indep_div_dep'))
f(x) = 2·x·atan(C1·x)
```

constant_renumber

`diofant.solvers.ode.constant_renumber(expr, symbolname, startnumber, endnumber)`

Renumber arbitrary constants in `expr` to have numbers 1 through N where N is `endnumber - startnumber + 1` at most. In the process, this reorders expression terms in a standard way.

This is a simple function that goes through and rennumbers any [Symbol](#) (page 80) 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 645).

Examples

```
>>> C0, C1, C2, C3, C4 = symbols('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)
C1·x + C2 + C3·x2
>>> pprint(constant_renumber(C2 + C1*x + C3*x**2, 'C', 1, 3))
C1 + C2·x + C3·x2
```

constantsimp

`diofant.solvers.ode.constantsimp(expr, constants)`

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with [dsolve\(\)](#) (page 635), and is not intended for general use.

Simplification is done by “absorbing” the arbitrary constants into other arbitrary constants, numbers, and symbols that they are not independent of.

The symbols must all have the same name with numbers after it, for example, 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 of constants is done with limited assistance:

1. terms of [Add](#) (page 104)s are collected to try join constants so $e^x(C_1 \cos(x) + C_2 \cos(x))$ will simplify to $e^x C_1 \cos(x)$;
2. powers with exponents that are [Add](#) (page 104)s are expanded so e^{C_1+x} will be simplified to $C_1 e^x$.

Use [constant_renumber\(\)](#) (page 644) 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 635).

Examples

```
>>> C1, C2, C3 = symbols('C1, C2, C3')
>>> constantsimp(2*C1*x, {C1, C2, C3})
C1*x
>>> constantsimp(C1 + 2 + x, {C1, C2, C3})
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, {C1, C2, C3})
C1 + C3*x
```

sol_simplicity

`diofant.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 405)s, this will automatically be considered less simple than any of the above.

This function returns an integer such that if solution A is simpler than solution B by above metric, then `ode_sol_simplicity(sola, func) < ode_sol_simplicity(solb, func)`.

Currently, the following are the numbers returned, but if the heuristic is ever improved, this may change. Only the ordering is guaranteed.

Simplicity	Return
<code>sol</code> solved for <code>func</code>	-2
<code>sol</code> not solved for <code>func</code> but can be	-1
<code>sol</code> is not solved nor solvable for <code>func</code>	<code>len(str(sol))</code>
<code>sol</code> contains an <i>Integral</i> (page 405)	<code>oo</code>

`oo` here means the Diofant infinity, which should compare greater than any integer.

If you already know `solve()` (page 606) cannot solve `sol`, you can use `trysolving=False` to skip that step, which is the only potentially slow step. For example, `dsolve()` (page 635) 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)))`.

```
>>> C1, C2 = symbols('C1, C2')

>>> 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))
00
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x) + f(x)), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)
```

1st_exact

`diofant.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:

```
>>> x0, y0, C1 = symbols('x0 y0 C1')
>>> P, Q, F = map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
...           Integral(Q(x0, t), (t, y0, y))), C1))
```

$$F(x, y) = \int_{x_0}^x P(t, y) dt + \int_{y_0}^y Q(x_0, t) dt = C_1$$

Where the first partials of P and Q exist and are continuous in a simply connected region.

A note: Diofant 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.

Examples

```
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
...        f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

References

- https://en.wikipedia.org/wiki/Exact_differential_equation
- [TP63], pp. 73.

1st_homogeneous_coeff_best

`diofant.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()` (page 646).

See the `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 650) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 649) doc-strings for more information on these hints. Note that there is no `ode_1st_homogeneous_coeff_best_Integral` hint.

Examples

```
>>> 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  $\left( \frac{3 \cdot x}{f(x)} + 1 \right)$ 
log(f(x)) = log(C1) -  $\frac{2}{3}$ 
```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- [TP63], pp. 59.

1st_homogeneous_coeff_subs_dep_div_indep

`diofant.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 $u_1 = \frac{\langle \text{dependent variable} \rangle}{\langle \text{independent variable} \rangle}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of [homogeneous_order\(\)](#) (page 641).

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 $y = u_1x$ (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:

```
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform)
g( $\frac{f(x)}{x}$ ) + h( $\frac{f(x)}{x}$ )  $\cdot \frac{d}{dx}(f(x))$ 
>>> pprint(dsolve(genform, f(x),
...               hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'))
...

$$\log(x) = C_1 + \int \frac{-h(u_1)}{u_1 \cdot h(u_1) + g(u_1)} d(u_1)$$

```

Where $u_1 h(u_1) + g(u_1) \neq 0$ and $x \neq 0$.

See also:

[diofant.solvers.ode.ode_1st_homogeneous_coeff_best](#) (page 648), [diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep](#) (page 650)

Examples

```
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
...             hint='1st_homogeneous_coeff_subs_dep_div_indep',
...             simplify=False))
...

$$\log\left(\frac{3 \cdot f(x)}{x} + \frac{f^3(x)}{x}\right)$$


$$\log(x) = \log(C_1) - \frac{3}{3}$$

```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- [TP63], pp. 59.

1st_homogeneous_coeff_subs_indep_div_dep

`diofant.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 641).

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:

```
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform)
g( $\frac{x}{f(x)}$ ) + h( $\frac{x}{f(x)}$ )  $\frac{d}{dx}$ (f(x))
>>> pprint(dsolve(genform, f(x),
...               hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'))
...

$$f(x) = e^{\int \frac{-g(u_2)}{u_2 \cdot g(u_2) + h(u_2)} d(u_2)} \cdot C_1$$

```

Where $u_2 g(u_2) + h(u_2) \neq 0$ and $f(x) \neq 0$.

See also:

`diofant.solvers.ode.ode_1st_homogeneous_coeff_best` (page 648), `diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep` (page 649)

Examples

```
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
...               hint='1st_homogeneous_coeff_subs_indep_div_dep',
...               simplify=False))
log(
  2
  3·x
  log(----- + 1)
  2
  f (x)
)
log(f(x)) = log(C1) - -----
                        3
```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- [TP63], pp. 59.

1st_linear

`diofant.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:

```
>>> P, Q = map(Function, ['P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x))
>>> pprint(genform)
P(x)·f(x) +  $\frac{d}{dx}$ (f(x)) = Q(x)
>>> pprint(dsolve(genform, f(x), hint='1st_linear_Integral'))
f(x) = e $-\int P(x) dx$  ·  $\left( C_1 + \int e^{\int P(x) dx} \cdot Q(x) dx \right)$ 
```

Examples

```
>>> pprint(dsolve(Eq(x*diff(f(x), x) - f(x), x**2*sin(x)),
...               f(x), '1st_linear'))
f(x) = x·(C1 - cos(x))
```

References

- https://en.wikipedia.org/wiki/Linear_differential_equation#First-order_equation_with_variable_coefficients
- [TP63], pp. 92.

Bernoulli

`diofant.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 651)). The general solution is:

```
>>> P, Q = map(Function, ['P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)
>>> pprint(genform)
P(x)·f(x) +  $\frac{d}{dx}(f(x)) = Q(x) \cdot f^n(x)$ 
>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'), wrap_line=False)
```

$$f(x) = \left(e^{-(-n+1) \cdot \int P(x) dx} \cdot \left(C_1 + (n-1) \cdot \int -e^{(-n+1) \cdot \int P(x) dx} \cdot Q(x) dx \right) \right)^{\frac{1}{-n+1}}$$

Note that the equation is separable when $n = 1$ (see the docstring of `ode_separable()` (page 657)).

```
>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x),
hint='separable_Integral'))

$$\int \frac{1}{y} dy = C_1 + \int (-P(x) + Q(x)) dx$$

```

Examples

```
>>> pprint(dsolve(Eq(x*f(x).diff(x) + f(x), log(x)*f(x)**2),
f(x), hint='Bernoulli'))

$$f(x) = \frac{1}{x \cdot \left( C_1 + \frac{\log(x)}{x} + \frac{1}{x} \right)}$$

```


References

- https://en.wikipedia.org/wiki/Bernoulli_differential_equation
- [TP63], pp. 95.

Liouville

`diofant.solvers.ode.ode_Liouville(eq, func, order, match)`

Solves 2nd order Liouville differential equations.

The general form of a Liouville ODE is

$$\frac{d^2 y}{dx^2} + g(y) \left(\frac{dy}{dx} \right)^2 + h(x) \frac{dy}{dx}.$$

The general solution is:

```
>>> genform = Eq(diff(f(x), x, x) + g(f(x))*diff(f(x), x)**2 +
...              h(x)*diff(f(x), x), 0)
>>> pprint(genform)
g(f(x)) * (d(f(x))/dx)**2 + h(x) * d(f(x))/dx + d(f(x))/dx**2 = 0
>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))
C1 + C2 * e^(- integral h(x) dx) + integral g(y) dy = 0
```

Examples

```
>>> pprint(dsolve(diff(f(x), x, x) + diff(f(x), x)**2/f(x) +
...              diff(f(x), x)/x, f(x), hint='Liouville'))
[f(x) = -sqrt(C1 + C2*log(x)), f(x) = sqrt(C1 + C2*log(x))]
```

References

- [GB73], pp. 98.
- <https://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Liouville>

Riccati_special_minus2

`diofant.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.

```
>>> genform = a*f(x).diff(x) - (b*f(x)**2 + c*f(x)/x + d/x**2)
>>> sol = dsolve(genform, f(x))
>>> pprint(sol, wrap_line=False)
```

$$f(x) = \frac{-\left(a + c - \sqrt{4 \cdot b \cdot d - (a + c)^2} \cdot \tan\left(C_1 + \frac{\sqrt{4 \cdot b \cdot d - (a + c)^2} \cdot \log(x)}{2 \cdot a}\right)\right)}{2 \cdot b \cdot x}$$

References

1. <https://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Riccati>
2. <http://eqworld.ipmnet.ru/en/solutions/ode/ode0106.pdf> - <http://eqworld.ipmnet.ru/en/solutions/ode/ode0123.pdf>

nth_linear_constant_coeff_homogeneous

`diofant.solvers.ode.ode_nth_linear_constant_coeff_homogeneous(eq, func, order, match, returns='sol')`

Solves an n th order linear homogeneous differential equation with constant coefficients.

This is an equation of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0.$$

These equations can be solved in a general manner, by taking the roots of the characteristic equation $a_n m^n + a_{n-1} m^{n-1} + \cdots + a_1 m + a_0 = 0$. The solution will then be the sum of $C_n x^i e^{rx}$ terms, for each where C_n is an arbitrary constant, r is a root of the characteristic equation and i is one of each from 0 to the multiplicity of the root - 1 (for example, a root 3 of multiplicity 2 would create the terms $C_1 e^{3x} + C_2 x e^{3x}$). The exponential is usually expanded for complex roots using Euler’s equation $e^{Ix} = \cos(x) + I \sin(x)$. Complex roots always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as $e^{ax} (C_1 \cos(bx) + C_2 \sin(bx))$.

If Diofant cannot find exact roots to the characteristic equation, a `RootOf` (page 540) instance will be return instead.

```
>>> dsolve(f(x).diff((x, 5)) + 10*f(x).diff(x) - 2*f(x), f(x),
...         hint='nth_linear_constant_coeff_homogeneous')
Eq(f(x), E**(x*RootOf(_x**5 + 10*_x - 2, 0))*C1 +
E**(x*RootOf(_x**5 + 10*_x - 2, 1))*C2 +
E**(x*RootOf(_x**5 + 10*_x - 2, 2))*C3 +
E**(x*RootOf(_x**5 + 10*_x - 2, 3))*C4 +
E**(x*RootOf(_x**5 + 10*_x - 2, 4))*C5)
```

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>}`.

Examples

```
>>> 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'))
f(x) = ex · (C3 + C4 · x) + e-2 · x · (C1 · sin(x) + C2 · cos(x))
```

References

- https://en.wikipedia.org/wiki/Linear_differential_equation section: Nonhomogeneous equation with constant coefficients
- [TP63], pp. 211.

nth_linear_constant_coeff_undetermined_coefficients

`diofant.solvers.ode.ode_nth_linear_constant_coeff_undetermined_coefficients`(*eq*, *func*, *or*, *der*, *match*)

Solves an n th order linear differential equation with constant coefficients using the method of undetermined coefficients.

This method works on differential equations of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x),$$

where $P(x)$ is a function that has a finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form $ax^i e^{bx} \sin(cx + d)$ or $ax^i e^{bx} \cos(cx + d)$, where i is a non-negative integer and a , b , c , and d are constants. For example any polynomial in x , functions like $x^2 e^{2x}$, $x \sin(x)$, and $e^x \cos(x)$ can all be used. Products of sin's and cos's have a finite number of derivatives, because they can be expanded into $\sin(ax)$ and $\cos(bx)$ terms. However, Diofant 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.

Examples

```
>>> pprint(dsolve(f(x).diff((x, 2)) + 2*f(x).diff(x) + f(x) -
...               4*exp(-x)*x**2 + cos(2*x), f(x),
...               hint='nth_linear_constant_coeff_undetermined_coefficients'))
```

$$f(x) = -\frac{4 \cdot \sin(2 \cdot x)}{25} + \frac{3 \cdot \cos(2 \cdot x)}{25} + e^{-x} \cdot \left(C_1 + C_2 \cdot x + \frac{x^4}{3} \right)$$

References

- https://en.wikipedia.org/wiki/Method_of_undetermined_coefficients
- [TP63], pp. 221.

nth_linear_constant_coeff_variation_of_parameters

`diofant.solvers.ode.ode_nth_linear_constant_coeff_variation_of_parameters(eq, func, or-der, match)`

Solves an n th order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1f'(x) + a_0f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{i=1}^n c_i(x) y_i(x),$$

where y_i is the i th solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by

$$\sum_{i=1}^n \left(\int \frac{W_i(x)}{W(x)} dx \right) y_i(x),$$

where $W(x)$ is the Wronskian of the fundamental system (the system of n linearly independent solutions to the homogeneous equation), and $W_i(x)$ is the Wronskian of the fundamental system with the i th column replaced with $[0, 0, \dots, 0, P(x)]$.

This method is general enough to solve any n th order inhomogeneous linear differential equation with constant coefficients, but sometimes Diofant 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 635) 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.

Examples

```
>>> pprint(dsolve(f(x).diff((x, 3)) - 3*f(x).diff((x, 2)) +
...             3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
...             hint='nth_linear_constant_coeff_variation_of_parameters'))
f(x) = ex ·  $\left( C_1 + C_2 \cdot x + C_3 \cdot x^2 + \frac{x^3 \cdot (6 \cdot \log(x) - 11)}{36} \right)$ 
```

References

- https://en.wikipedia.org/wiki/Variation_of_parameters
- [TP63], pp. 233.

separable

`diofant.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 `diofant.simplify.simplify.separatevars()` (page 582) as its back end, so if a separable equation is not caught by this solver, it is most likely the fault of that function. `separatevars()` (page 582) 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:

```
>>> a, b, c, d = map(Function, ['a', 'b', 'c', 'd'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform)
a(x)·b(f(x))· $\frac{d}{dx}(f(x)) = c(x)·d(f(x))$ 
>>> pprint(dsolve(genform, f(x), hint='separable_Integral'))
f(x)

$$\int \frac{b(y)}{d(y)} dy = C_1 + \int \frac{c(x)}{a(x)} dx$$

```

Examples

```
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x) + x, 3*x*f(x)**2), f(x),
...               hint='separable', simplify=False))

$$\frac{\log\left(\frac{2}{3 \cdot f(x)^2} - 1\right)}{6} = C_1 + \frac{x^2}{2}$$

```

References

- [TP63], pp. 52.

almost_linear

diofant.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

```
>>> k, l = map(Function, ['k', 'l'])
>>> genform = Eq(f(x)*(l(y).diff(y)) + k(x)*l(y) + g(x), 0)
>>> pprint(genform)
f(x)· $\frac{d}{dy}(l(y)) + g(x) + k(x)·l(y) = 0$ 
>>> pprint(dsolve(genform, hint='almost_linear'))

$$l(y) = e^{\int \frac{-y \cdot k(x)}{f(x)} dx} \cdot \left( C_1 + \int \begin{cases} \frac{-y \cdot g(x)}{f(x)} & \text{for } k(x) = 0 \\ \frac{y \cdot k(x)}{f(x)} & \text{otherwise} \end{cases} dx \right)$$

```

(continues on next page)

(continued from previous page)

$$\left(\left(\begin{array}{c} k(x) \end{array} \right) \right)$$

See also:

[`diofant.solvers.ode.ode_1st_linear\(\)`](#) (page 651)

Examples

```
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), E**(-x)*(C1 - Ei(x)))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'))
f(x) = e-x · (C1 - Ei(x))
```

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

linear_coefficients

`diofant.solvers.ode.ode_linear_coefficients(eq, func, order, match)`

Solves a differential equation with linear coefficients.

The general form of a differential equation with linear coefficients is

$$y' + F\left(\frac{a_1x + b_1y + c_1}{a_2x + b_2y + c_2}\right) = 0,$$

where $a_1, b_1, c_1, a_2, b_2, c_2$ are constants and $a_1b_2 - a_2b_1 \neq 0$.

This can be solved by substituting:

$$x = x' + \frac{b_2c_1 - b_1c_2}{a_2b_1 - a_1b_2}$$

$$y = y' + \frac{a_1c_2 - a_2c_1}{a_2b_1 - a_1b_2}.$$

This substitution reduces the equation to a homogeneous differential equation.

See also:

[`diofant.solvers.ode.ode_1st_homogeneous_coeff_best\(\)`](#) (page 648), [`diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep\(\)`](#) (page 650), [`diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep\(\)`](#) (page 649)

Examples

```
>>> eq = (x + f(x) + 1)*f(x).diff(x) + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[Eq(f(x), -x - sqrt(C1 + 7*x**2) - 1), Eq(f(x), -x + sqrt(C1 + 7*x**2) - 1)]
>>> pprint(dsolve(eq, hint='linear_coefficients'))
```

$$\left[f(x) = -x - \sqrt{C_1 + 7 \cdot x^2} - 1, f(x) = -x + \sqrt{C_1 + 7 \cdot x^2} - 1 \right]$$

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

separable_reduced

`diofant.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:

```
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform)
```

$$\frac{d}{dx}(f(x)) + \frac{f(x) \cdot g\left(x^n \cdot f(x)\right)}{x^n \cdot f(x)}$$

```
>>> pprint(dsolve(genform, hint='separable_reduced'))
```

$$\int \frac{1}{y \cdot (n - g(y))} dy = C_1 + \log(x)$$

See also:

[`diofant.solvers.ode.ode_separable\(\)`](#) (page 657)

Examples

```
>>> eq = (x - x**2*f(x))*f(x).diff(x) - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (-sqrt(C1*x**2 + 1) + 1)/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/x)]
>>> pprint(dsolve(eq, hint='separable_reduced'))
```

$$\left[f(x) = \frac{-\sqrt{C_1 \cdot x^2 + 1} + 1}{x}, f(x) = \frac{\sqrt{C_1 \cdot x^2 + 1} + 1}{x} \right]$$

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

lie_group

`diofant.solvers.ode.ode_lie_group(eq, func, order, match)`

This hint implements the Lie group method of solving first order differential equations. The aim is to convert the given differential equation from the given coordinate system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE is quadrature and can be solved easily. It makes use of the `diofant.solvers.ode.infinitesimals()` (page 641) function which returns the infinitesimals of the transformation.

The coordinates r and s can be found by solving the following Partial Differential Equations.

$$\xi \frac{\partial r}{\partial x} + \eta \frac{\partial r}{\partial y} = 0$$

$$\xi \frac{\partial s}{\partial x} + \eta \frac{\partial s}{\partial y} = 1$$

The differential equation becomes separable in the new coordinate system

$$\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x, y) \frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x, y) \frac{\partial r}{\partial y}}$$

After finding the solution by integration, it is then converted back to the original coordinate system by substituting r and s in terms of x and y again.

Examples

```
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
...               hint='lie_group'))
```

$$f(x) = e^{-x^2} \cdot \left(C_1 + \frac{x^2}{2} \right)$$

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1-14.

1st_power_series

`diofant.solvers.ode.ode_1st_power_series(eq, func, order, match)`

The power series solution is a method which gives the Taylor series expansion to the solution of a differential equation.

For a first order differential equation $\frac{dy}{dx} = h(x, y)$, a power series solution exists at a point $x = x_0$ if $h(x, y)$ is analytic at x_0 . The solution is given by

$$y(x) = y(x_0) + \sum_{n=1}^{\infty} \frac{F_n(x_0, b)(x - x_0)^n}{n!},$$

where $y(x_0) = b$ is the value of y at the initial value of x_0 . To compute the values of the $F_n(x_0, b)$ the following algorithm is followed, until the required number of terms are generated.

1. $F_1 = h(x_0, b)$
2. $F_{n+1} = \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1$

Examples

```
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'))
```

$$f(x) = C_1 + C_1 \cdot x - \frac{C_1 \cdot x^3}{6} + \frac{C_1 \cdot x^4}{24} + \frac{C_1 \cdot x^5}{60} + O\left(x^6\right)$$

References

- Travis W. Walker, Analytic power series technique for solving first-order differential equations, pp 17, 18

2nd_power_series_ordinary

`diofant.solvers.ode.ode_2nd_power_series_ordinary(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at an ordinary point. A homogenous differential equation is of the form

$$P(x) \frac{d^2 y}{dx^2} + Q(x) \frac{dy}{dx} + R(x)y = 0$$

For simplicity it is assumed that $P(x)$, $Q(x)$ and $R(x)$ are polynomials, it is sufficient that $\frac{Q(x)}{P(x)}$ and $\frac{R(x)}{P(x)}$ exists at x_0 . A recurrence relation is obtained by substituting y as $\sum_{n=0}^{\infty} a_n x^n$, in the differential equation, and equating the n th term. Using this relation various terms can be generated.

Examples

```
>>> eq = f(x).diff((x, 2)) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'))
```

$$f(x) = C_2 \cdot \left(\frac{x^4}{24} - \frac{x^2}{2} + 1 \right) + C_1 \cdot x \cdot \left(-\frac{x^2}{6} + 1 \right) + O\left(x^6\right)$$

References

- <http://tutorial.math.lamar.edu/Classes/DE/SeriesSolutions.aspx>
- [Sim16], pp 176 - 184.

2nd_power_series_regular

`diofant.solvers.ode.ode_2nd_power_series_regular(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at a regular point. A second order homogenous differential equation is of the form

$$P(x) \frac{d^2 y}{dx^2} + Q(x) \frac{dy}{dx} + R(x)y = 0$$

A point is said to regular singular at x_0 if $x - x_0 \frac{Q(x)}{P(x)}$ and $(x - x_0)^2 \frac{R(x)}{P(x)}$ are analytic at x_0 . For simplicity $P(x)$, $Q(x)$ and $R(x)$ are assumed to be polynomials. The algorithm for finding the power series solutions is:

1. Try expressing $(x - x_0)P(x)$ and $((x - x_0)^2)Q(x)$ as power series solutions about x_0 . Find p_0 and q_0 which are the constants of the power series expansions.
2. Solve the indicial equation $f(m) = m(m - 1) + m * p_0 + q_0$, to obtain the roots m_1 and m_2 of the indicial equation.
3. If $m_1 - m_2$ is a non integer there exists two series solutions. If $m_1 = m_2$, there exists only one solution. If $m_1 - m_2$ is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form $x^m \sum_{n=0}^{\infty} a_n x^n$. The coefficients are determined by the following recurrence relation. $a_n = -\frac{\sum_{k=0}^{n-1} q_{n-k} + (m+k)p_{n-k}}{f(m+n)}$. For the case in which $m_1 - m_2$ is an integer, it can be seen from the recurrence relation that for the lower root m , when n equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.

Examples

```
>>> eq = x*(f(x).diff((x, 2))) + 2*(f(x).diff(x)) + x*f(x)
>>> pprint(dsolve(eq))
```

$$f(x) = C_2 \cdot \left(\frac{x^4}{120} - \frac{x^2}{6} + 1 \right) + \frac{C_1 \cdot \left(-\frac{x^6}{720} + \frac{x^4}{24} - \frac{x^2}{2} + 1 \right)}{x} + O\left(\frac{1}{x^6}\right)$$

References

- [Sim16], pp 176 - 184.

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

`diofant.solvers.ode.lie_heuristic_abaco1_simple(match, comp)`

The first heuristic uses the following four sets of assumptions on ξ and η

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption $\xi = 0$ and η to be a function of x , the PDE

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi * \frac{\partial h}{\partial x} - \eta * \frac{\partial h}{\partial y} = 0$$

reduces to $f'(x) - f \frac{\partial h}{\partial y} = 0$. If $\frac{\partial h}{\partial y}$ is a function of x , then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco1_product

`diofant.solvers.ode.lie_heuristic_abaco1_product(match, comp)`

The second heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) * g(y)$$

$$\eta = f(x) * g(y), \xi = 0$$

The first assumption of this heuristic holds good if $\frac{1}{h^2} \frac{\partial^2}{\partial x \partial y} \log(h)$ is separable in x and y , then the separated factors containing x is $f(x)$, and $g(y)$ is obtained by

$$e^{\int f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right) dy}$$

provided $f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right)$ is a function of y only.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) * g(y)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

bivariate

`diofant.solvers.ode.lie_heuristic_bivariate(match, comp)`

The third heuristic assumes the infinitesimals ξ and η to be bi-variate polynomials in x and y . The assumption made here for the logic below is that h is a rational function in x and y though that may not be necessary for the infinitesimals to be bivariate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

References

- Lie Groups and Differential Equations pp. 327 - pp. 329

chi

`diofant.solvers.ode.lie_heuristic_chi(match, comp)`

The aim of the fourth heuristic is to find the function $\chi(x, y)$ that satisfies the PDE $\frac{d\chi}{dx} + h \frac{d\chi}{dy} - \frac{\partial h}{\partial y} \chi = 0$.

This assumes χ to be a bivariate polynomial in x and y . By intuition, h should be a rational function in x and y . The method used here is to substitute a general binomial for χ up to a

certain maximum degree is reached. The coefficients of the polynomials, are calculated by by collecting terms of the same order in x and y .

After finding χ , the next step is to use $\eta = \xi * h + \chi$, to determine ξ and η . This can be done by dividing χ by h which would give $-\xi$ as the quotient and η as the remainder.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A,C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco2_similar

`diofant.solvers.ode.lie_heuristic_abaco2_similar(match, comp)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = g(x), \xi = f(x)$$

$$\eta = f(y), \xi = g(y)$$

For the first assumption,

1. First $\frac{\frac{\partial h}{\partial y}}{\frac{\partial^2 h}{\partial y^2}}$ is calculated. Let us say this value is A
2. If this is constant, then h is matched to the form $A(x) + B(x)e^{\frac{y}{C}}$ then, $\frac{e^{\int \frac{A(x)}{C} dx}}{B(x)}$ gives $f(x)$ and $A(x) * f(x)$ gives $g(x)$
3. Otherwise $\frac{\frac{\partial A}{\partial X}}{\frac{\partial A}{\partial Y}} = \gamma$ is calculated. If
 - a] γ is a function of x alone
 - b] $\frac{\gamma \frac{\partial h}{\partial y} - \gamma'(x) - \frac{\partial h}{\partial x}}{h + \gamma} = G$ is a function of x alone. then, $e^{\int G dx}$ gives $f(x)$ and $-\gamma * f(x)$ gives $g(x)$

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(x)$, the coordinates are again interchanged, to get ξ as $f(x^*)$ and η as $g(y^*)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

function_sum

`diofant.solvers.ode.lie_heuristic_function_sum(match, comp)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) + g(y)$$

$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y} \left[\left(h \frac{\partial^2}{\partial x^2} (h^{-1}) \right)^{-1} \right]$$

is separable in x and y ,

1. The separated factors containing y is $\frac{\partial g}{\partial y}$. From this $g(y)$ can be determined.
2. The separated factors containing x is $f''(x)$.
3. $h \frac{\partial^2}{\partial x^2} (h^{-1})$ equals $\frac{f''(x)}{f(x)+g(y)}$. From this $f(x)$ can be determined.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) + g(y)$.

For both assumptions, the constant factors are separated among $g(y)$ and $f''(x)$, such that $f''(x)$ obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

abaco2_unique_unknown

`diofant.solvers.ode.lie_heuristic_abaco2_unique_unknown(match, comp)`

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1. A list of all functions and non-integer powers containing x and y
2. Loop over each element f in the list, find $\frac{\partial f}{\frac{\partial f}{\partial x}} = R$

If it is separable in x and y , let X be the factors containing x . Then

a] Check if $\xi = X$ and $\eta = -\frac{X}{R}$ satisfy the PDE. If yes, then return ξ and η

b] Check if $\xi = \frac{-R}{X}$ and $\eta = -\frac{1}{X}$ satisfy the PDE.
If yes, then return ξ and η

If not, then check if

a] $\xi = -R, \eta = 1$

b] $\xi = 1, \eta = -\frac{1}{R}$

are solutions.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

linear

`diofant.solvers.ode.lie_heuristic_linear(match, comp)`

This heuristic assumes

1. $\xi = ax + by + c$ and
2. $\eta = fx + gy + h$

After substituting the following assumptions in the determining PDE, it reduces to

$$f + (g - a)h - bh^2 - (ax + by + c)\frac{\partial h}{\partial x} - (fx + gy + c)\frac{\partial h}{\partial y}$$

Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that h need not be a rational function in this case.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

System of ODEs

These functions are intended for internal use by `dsolve()` (page 635) for system of differential equations.

system_of_odes_linear_2eq_order1_type3

`diofant.solvers.ode._linear_2eq_order1_type3(x, y, t, r, eq)`

The equations of this type of ode are

$$x' = f(t)x + g(t)y$$

$$y' = g(t)x + f(t)y$$

The solution of such equations is given by

$$x = e^F(C_1 e^G + C_2 e^{-G}), y = e^F(C_1 e^G - C_2 e^{-G})$$

where C_1 and C_2 are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

system_of_odes_linear_2eq_order1_type4

`diofant.solvers.ode._linear_2eq_order1_type4(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = -g(t)x + f(t)y$$

The solution is given by

$$x = F(C_1 \cos(G) + C_2 \sin(G)), y = F(-C_1 \sin(G) + C_2 \cos(G))$$

where C_1 and C_2 are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

system_of_odes_linear_2eq_order1_type5

`diofant.solvers.ode._linear_2eq_order1_type5(x, y, t, r, eq)`

The equations of this type of ode are

$$x' = f(t)x + g(t)y$$

$$y' = ag(t)x + [f(t) + bg(t)]y$$

The transformation of

$$x = e^{\int f(t) dt} u, y = e^{\int f(t) dt} v, T = \int g(t) dt$$

leads to a system of constant coefficient linear differential equations

$$u'(T) = v, v'(T) = au + bv$$

system_of_odes_linear_2eq_order1_type6

`diofant.solvers.ode._linear_2eq_order1_type6(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = a[f(t) + ah(t)]x + a[g(t) - h(t)]y$$

This is solved by first multiplying the first equation by $-a$ and adding it to the second equation to obtain

$$y' - ax' = -ah(t)(y - ax)$$

Setting $U = y - ax$ and integrating the equation we arrive at

$$y - ax = C_1 e^{-a \int h(t) dt}$$

and on substituting the value of y in first equation give rise to first order ODEs. After solving for x , we can obtain y by substituting the value of x in second equation.

system_of_odes_linear_2eq_order1_type7

`diofant.solvers.ode._linear_2eq_order1_type7(x, y, t, r, eq)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = h(t)x + p(t)y$$

Differentiating the first equation and substituting the value of y from second equation will give a second-order linear equation

$$gx'' - (fg + gp + g')x' + (fgp - g^2h + fg' - f'g)x = 0$$

This above equation can be easily integrated if following conditions are satisfied.

1. $fgp - g^2h + fg' - f'g = 0$
2. $fgp - g^2h + fg' - f'g = ag, fg + gp + g' = bg$

If first condition is satisfied then it is solved by current dsolve solver and in second case it becomes a constant coefficient differential equation which is also solved by current solver.

Otherwise if the above condition fails then, a particular solution is assumed as $x = x_0(t)$ and $y = y_0(t)$ Then the general solution is expressed as

$$x = C_1 x_0(t) + C_2 x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt$$

$$y = C_1 y_0(t) + C_2 \left[\frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt \right]$$

where C_1 and C_2 are arbitrary constants and

$$F(t) = e^{\int f(t) dt}, P(t) = e^{\int p(t) dt}$$

system_of_odes_linear_2eq_order2_type1

`diofant.solvers.ode._linear_2eq_order2_type1(x, y, t, r, eq)`

System of two constant-coefficient second-order linear homogeneous differential equations

$$x'' = ax + by$$

$$y'' = cx + dy$$

The characteristic equation for above equations

$$\lambda^4 - (a + d)\lambda^2 + ad - bc = 0$$

whose discriminant is $D = (a - d)^2 + 4bc \neq 0$

1. When $ad - bc \neq 0$

1.1. If $D \neq 0$. The characteristic equation has four distinct roots, $\lambda_1, \lambda_2, \lambda_3, \lambda_4$. The general solution of the system is

$$x = C_1 b e^{\lambda_1 t} + C_2 b e^{\lambda_2 t} + C_3 b e^{\lambda_3 t} + C_4 b e^{\lambda_4 t}$$

$$y = C_1 (\lambda_1^2 - a) e^{\lambda_1 t} + C_2 (\lambda_2^2 - a) e^{\lambda_2 t} + C_3 (\lambda_3^2 - a) e^{\lambda_3 t} + C_4 (\lambda_4^2 - a) e^{\lambda_4 t}$$

where C_1, \dots, C_4 are arbitrary constants.

1.2. If $D = 0$ and $a \neq d$:

$$x = 2C_1 \left(bt + \frac{2bk}{a-d} \right) e^{\frac{kt}{2}} + 2C_2 \left(bt + \frac{2bk}{a-d} \right) e^{-\frac{kt}{2}} + 2bC_3 t e^{\frac{kt}{2}} + 2bC_4 t e^{-\frac{kt}{2}}$$

$$y = C_1 (d-a) t e^{\frac{kt}{2}} + C_2 (d-a) t e^{-\frac{kt}{2}} + C_3 [(d-a)t + 2k] e^{\frac{kt}{2}} + C_4 [(d-a)t - 2k] e^{-\frac{kt}{2}}$$

where C_1, \dots, C_4 are arbitrary constants and $k = \sqrt{2(a+d)}$

1.3. If $D = 0$ and $a = d \neq 0$ and $b = 0$:

$$x = 2\sqrt{a}C_1 e^{\sqrt{a}t} + 2\sqrt{a}C_2 e^{-\sqrt{a}t}$$

$$y = cC_1 t e^{\sqrt{a}t} - cC_2 t e^{-\sqrt{a}t} + C_3 e^{\sqrt{a}t} + C_4 e^{-\sqrt{a}t}$$

1.4. If $D = 0$ and $a = d \neq 0$ and $c = 0$:

$$x = bC_1 t e^{\sqrt{a}t} - bC_2 t e^{-\sqrt{a}t} + C_3 e^{\sqrt{a}t} + C_4 e^{-\sqrt{a}t}$$

$$y = 2\sqrt{a}C_1 e^{\sqrt{a}t} + 2\sqrt{a}C_2 e^{-\sqrt{a}t}$$

2. When $ad - bc = 0$ and $a^2 + b^2 > 0$. Then the original system becomes

$$x'' = ax + by$$

$$y'' = k(ax + by)$$

2.1. If $a + bk \neq 0$:

$$x = C_1 e^{t\sqrt{a+bk}} + C_2 e^{-t\sqrt{a+bk}} + C_3 bt + C_4 b$$

$$y = C_1 k e^{t\sqrt{a+bk}} + C_2 k e^{-t\sqrt{a+bk}} - C_3 at - C_4 a$$

2.2. If $a + bk = 0$:

$$x = C_1 bt^3 + C_2 bt^2 + C_3 t + C_4$$

$$y = kx + 6C_1 t + 2C_2$$

system_of_odes_linear_2eq_order2_type2

`diofant.solvers.ode._linear_2eq_order2_type2(x, y, t, r, eq)`

The equations in this type are

$$x'' = a_1x + b_1y + c_1$$

$$y'' = a_2x + b_2y + c_2$$

The general solution of this system is given by the sum of its particular solution and the general solution of the homogeneous system. The general solution is given by the linear system of 2 equation of order 2 and type 1

1. If $a_1b_2 - a_2b_1 \neq 0$. A particular solution will be $x = x_0$ and $y = y_0$ where the constants x_0 and y_0 are determined by solving the linear algebraic system

$$a_1x_0 + b_1y_0 + c_1 = 0, a_2x_0 + b_2y_0 + c_2 = 0$$

2. If $a_1b_2 - a_2b_1 = 0$ and $a_1^2 + b_1^2 > 0$. In this case, the system in question becomes

$$x'' = ax + by + c_1, y'' = k(ax + by) + c_2$$

2.1. If $\sigma = a + bk \neq 0$, the particular solution will be

$$x = \frac{1}{2}b\sigma^{-1}(c_1k - c_2)t^2 - \sigma^{-2}(ac_1 + bc_2)$$

$$y = kx + \frac{1}{2}(c_2 - c_1k)t^2$$

2.2. If $\sigma = a + bk = 0$, the particular solution will be

$$x = \frac{1}{24}b(c_2 - c_1k)t^4 + \frac{1}{2}c_1t^2$$

$$y = kx + \frac{1}{2}(c_2 - c_1k)t^2$$

system_of_odes_linear_2eq_order2_type3

`diofant.solvers.ode._linear_2eq_order2_type3(x, y, t, r, eq)`

These type of equation is used for describing the horizontal motion of a pendulum taking into account the Earth rotation. The solution is given with $a^2 + 4b > 0$:

$$x = C_1 \cos(\alpha t) + C_2 \sin(\alpha t) + C_3 \cos(\beta t) + C_4 \sin(\beta t)$$

$$y = -C_1 \sin(\alpha t) + C_2 \cos(\alpha t) - C_3 \sin(\beta t) + C_4 \cos(\beta t)$$

where C_1, \dots, C_4 and

$$\alpha = \frac{1}{2}a + \frac{1}{2}\sqrt{a^2 + 4b}, \beta = \frac{1}{2}a - \frac{1}{2}\sqrt{a^2 + 4b}$$

system_of_odes_linear_2eq_order2_type5

`diofant.solvers.ode._linear_2eq_order2_type5(x, y, t, r, eq)`

The equation which come under this category are

$$x'' = a(ty' - y)$$

$$y'' = b(tx' - x)$$

The transformation

$$u = tx' - x, b = ty' - y$$

leads to the first-order system

$$u' = atv, v' = btu$$

The general solution of this system is given by

If $ab > 0$:

$$u = C_1 a e^{\frac{1}{2}\sqrt{ab}t^2} + C_2 a e^{-\frac{1}{2}\sqrt{ab}t^2}$$

$$v = C_1 \sqrt{ab} e^{\frac{1}{2}\sqrt{ab}t^2} - C_2 \sqrt{ab} e^{-\frac{1}{2}\sqrt{ab}t^2}$$

If $ab < 0$:

$$u = C_1 a \cos\left(\frac{1}{2}\sqrt{|ab|}t^2\right) + C_2 a \sin\left(-\frac{1}{2}\sqrt{|ab|}t^2\right)$$

$$v = C_1 \sqrt{|ab|} \sin\left(\frac{1}{2}\sqrt{|ab|}t^2\right) + C_2 \sqrt{|ab|} \cos\left(-\frac{1}{2}\sqrt{|ab|}t^2\right)$$

where C_1 and C_2 are arbitrary constants. On substituting the value of u and v in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{v}{t^2} dt$$

where C_3 and C_4 are arbitrary constants.

system_of_odes_linear_2eq_order2_type6

`diofant.solvers.ode._linear_2eq_order2_type6(x, y, t, r, eq)`

The equations are

$$x'' = f(t)(a_1 x + b_1 y)$$

$$y'' = f(t)(a_2 x + b_2 y)$$

If k_1 and k_2 are roots of the quadratic equation

$$k^2 - (a_1 + b_2)k + a_1 b_2 - a_2 b_1 = 0$$

Then by multiplying appropriate constants and adding together original equations we obtain two independent equations:

$$z_1'' = k_1 f(t) z_1, z_1 = a_2 x + (k_1 - a_1) y$$

$$z_2'' = k_2 f(t) z_2, z_2 = a_2 x + (k_2 - a_1) y$$

Solving the equations will give the values of x and y after obtaining the value of z_1 and z_2 by solving the differential equation and substituting the result.

system_of_odes_linear_2eq_order2_type7

`diofant.solvers.ode._linear_2eq_order2_type7(x, y, t, r, eq)`

The equations are given as

$$x'' = f(t)(a_1x' + b_1y')$$

$$y'' = f(t)(a_2x' + b_2y')$$

If k_1 and ' k_2 ' are roots of the quadratic equation

$$k^2 - (a_1 + b_2)k + a_1b_2 - a_2b_1 = 0$$

Then the system can be reduced by adding together the two equations multiplied by appropriate constants give following two independent equations:

$$z_1'' = k_1f(t)z_1', z_1 = a_2x + (k_1 - a_1)y$$

$$z_2'' = k_2f(t)z_2', z_2 = a_2x + (k_2 - a_1)y$$

Integrating these and returning to the original variables, one arrives at a linear algebraic system for the unknowns x and y :

$$a_2x + (k_1 - a_1)y = C_1 \int e^{k_1 F(t)} dt + C_2$$

$$a_2x + (k_2 - a_1)y = C_3 \int e^{k_2 F(t)} dt + C_4$$

where C_1, \dots, C_4 are arbitrary constants and $F(t) = \int f(t) dt$

system_of_odes_linear_2eq_order2_type8

`diofant.solvers.ode._linear_2eq_order2_type8(x, y, t, r, eq)`

The equation of this category are

$$x'' = af(t)(ty' - y)$$

$$y'' = bf(t)(tx' - x)$$

The transformation

$$u = tx' - x, v = ty' - y$$

leads to the system of first-order equations

$$u' = atf(t)v, v' = bt f(t)u$$

The general solution of this system has the form

If $ab > 0$:

$$u = C_1 a e^{\sqrt{ab} \int t f(t) dt} + C_2 a e^{-\sqrt{ab} \int t f(t) dt}$$

$$v = C_1 \sqrt{ab} e^{\sqrt{ab} \int t f(t) dt} - C_2 \sqrt{ab} e^{-\sqrt{ab} \int t f(t) dt}$$

If $ab < 0$:

$$u = C_1 a \cos(\sqrt{|ab|} \int t f(t) dt) + C_2 a \sin(-\sqrt{|ab|} \int t f(t) dt)$$

$$v = C_1 \sqrt{|ab|} \sin(\sqrt{|ab|} \int t f(t) dt) + C_2 \sqrt{|ab|} \cos(-\sqrt{|ab|} \int t f(t) dt)$$

where C_1 and C_2 are arbitrary constants. On substituting the value of u and v in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{v}{t^2} dt$$

where C_3 and C_4 are arbitrary constants.

system_of_odes_linear_2eq_order2_type9

`diofant.solvers.ode._linear_2eq_order2_type9(x, y, t, r, eq)`

$$t^2 x'' + a_1 t x' + b_1 t y' + c_1 x + d_1 y = 0$$

$$t^2 y'' + a_2 t x' + b_2 t y' + c_2 x + d_2 y = 0$$

These system of equations are euler type.

The substitution of $t = \sigma e^\tau$ ($\sigma \neq 0$) leads to the system of constant coefficient linear differential equations

$$x'' + (a_1 - 1)x' + b_1 y' + c_1 x + d_1 y = 0$$

$$y'' + a_2 x' + (b_2 - 1)y' + c_2 x + d_2 y = 0$$

The general solution of the homogeneous system of differential equations is determined by a linear combination of linearly independent particular solutions determined by the method of undetermined coefficients in the form of exponentials

$$x = A e^{\lambda t}, y = B e^{\lambda t}$$

On substituting these expressions into the original system and collecting the coefficients of the unknown A and B , one obtains

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)A + (b_1\lambda + d_1)B = 0$$

$$(a_2\lambda + c_2)A + (\lambda^2 + (b_2 - 1)\lambda + d_2)B = 0$$

The determinant of this system must vanish for nontrivial solutions A, B to exist. This requirement results in the following characteristic equation for λ

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)(\lambda^2 + (b_2 - 1)\lambda + d_2) - (b_1\lambda + d_1)(a_2\lambda + c_2) = 0$$

If all roots k_1, \dots, k_4 of this equation are distinct, the general solution of the original system of the differential equations has the form

$$x = C_1(b_1\lambda_1 + d_1)e^{\lambda_1 t} - C_2(b_1\lambda_2 + d_1)e^{\lambda_2 t} - C_3(b_1\lambda_3 + d_1)e^{\lambda_3 t} - C_4(b_1\lambda_4 + d_1)e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 + (a_1 - 1)\lambda_1 + c_1)e^{\lambda_1 t} + C_2(\lambda_2^2 + (a_1 - 1)\lambda_2 + c_1)e^{\lambda_2 t} + C_3(\lambda_3^2 + (a_1 - 1)\lambda_3 + c_1)e^{\lambda_3 t} + C_4(\lambda_4^2 + (a_1 - 1)\lambda_4 + c_1)e^{\lambda_4 t}$$

system_of_odes_linear_2eq_order2_type11

`diofant.solvers.ode._linear_2eq_order2_type11(x, y, t, r, eq)`

The equations which comes under this type are

$$x'' = f(t)(tx' - x) + g(t)(ty' - y)$$

$$y'' = h(t)(tx' - x) + p(t)(ty' - y)$$

The transformation

$$u = tx' - x, v = ty' - y$$

leads to the linear system of first-order equations

$$u' = tf(t)u + tg(t)v, v' = th(t)u + tp(t)v$$

On substituting the value of u and v in transformed equation gives value of x and y as

$$x = C_3t + t \int \frac{u}{t^2} dt, y = C_4t + t \int \frac{v}{t^2} dt.$$

where C_3 and C_4 are arbitrary constants.

system_of_odes_linear_3eq_order1_type4

`diofant.solvers.ode._linear_3eq_order1_type4(x, y, z, t, r, eq)`

Equations:

$$x' = (a_1f(t) + g(t))x + a_2f(t)y + a_3f(t)z$$

$$y' = b_1f(t)x + (b_2f(t) + g(t))y + b_3f(t)z$$

$$z' = c_1f(t)x + c_2f(t)y + (c_3f(t) + g(t))z$$

The transformation

$$x = e^{\int g(t) dt}u, y = e^{\int g(t) dt}v, z = e^{\int g(t) dt}w, \tau = \int f(t) dt$$

leads to the system of constant coefficient linear differential equations

$$u' = a_1u + a_2v + a_3w$$

$$v' = b_1u + b_2v + b_3w$$

$$w' = c_1u + c_2v + c_3w$$

These system of equations are solved by homogeneous linear system of constant coefficients of n equations of first order. Then substituting the value of u, v and w in transformed equation gives value of x, y and z .

system_of_odes_linear_neq_order1_type1

`diofant.solvers.ode.sysode_linear_neq_order1(match_)`

System of n first-order constant-coefficient linear differential equations

$$Mx' = Lx + f(t)$$

Notes

Mass-matrix assumed to be invertible and provided general solution uses the Jordan canonical form for $A = M^{-1}L$.

References

- [HNorsettW14], pp. 73-76.

system_of_odes_nonlinear_2eq_order1_type1

`diofant.solvers.ode._nonlinear_2eq_order1_type1(x, y, t, eq)`

Equations:

$$x' = x^n F(x, y)$$

$$y' = g(y) F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if $n \neq 1$

$$\varphi = [C_1 + (1 - n) \int \frac{1}{g(y)} dy]^{\frac{1}{1-n}}$$

if $n = 1$

$$\varphi = C_1 e^{\int \frac{1}{g(y)} dy}$$

where C_1 and C_2 are arbitrary constants.

system_of_odes_nonlinear_2eq_order1_type2

`diofant.solvers.ode._nonlinear_2eq_order1_type2(x, y, t, eq)`

Equations:

$$x' = e^{\lambda x} F(x, y)$$

$$y' = g(y) F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if $\lambda \neq 0$

$$\varphi = -\frac{1}{\lambda} \log(C_1 - \lambda \int \frac{1}{g(y)} dy)$$

if $\lambda = 0$

$$\varphi = C_1 + \int \frac{1}{g(y)} dy$$

where C_1 and C_2 are arbitrary constants.

system_of_odes_nonlinear_2eq_order1_type3

`diofant.solvers.ode._nonlinear_2eq_order1_type3(x, y, t, eq)`

Autonomous system of general form

$$x' = F(x, y)$$

$$y' = G(x, y)$$

Assuming $y = y(x, C_1)$ where C_1 is an arbitrary constant is the general solution of the first-order equation

$$F(x, y)y'_x = G(x, y)$$

Then the general solution of the original system of equations has the form

$$\int \frac{1}{F(x, y(x, C_1))} dx = t + C_1$$

system_of_odes_nonlinear_2eq_order1_type4

`diofant.solvers.ode._nonlinear_2eq_order1_type4(x, y, t, eq)`

Equation:

$$x' = f_1(x)g_1(y)\phi(x, y, t)$$

$$y' = f_2(x)g_2(y)\phi(x, y, t)$$

First integral:

$$\int \frac{f_2(x)}{f_1(x)} dx - \int \frac{g_1(y)}{g_2(y)} dy = C$$

where C is an arbitrary constant.

On solving the first integral for x (resp., y) and on substituting the resulting expression into either equation of the original solution, one arrives at a first-order equation for determining y (resp., x).

system_of_odes_nonlinear_2eq_order1_type5

`diofant.solvers.ode._nonlinear_2eq_order1_type5(func, t, eq)`

Clairaut system of ODEs

$$x = tx' + F(x', y')$$

$$y = ty' + G(x', y')$$

The following are solutions of the system

(i) straight lines:

$$x = C_1 t + F(C_1, C_2), y = C_2 t + G(C_1, C_2)$$

where C_1 and C_2 are arbitrary constants;

(ii) envelopes of the above lines;

(iii) continuously differentiable lines made up from segments of the lines (i) and (ii).

system_of_odes_nonlinear_3eq_order1_type1

`diofant.solvers.ode._nonlinear_3eq_order1_type1(x, y, z, t, eq)`

Equations:

$$ax' = (b - c)yz, \quad by' = (c - a)zx, \quad cz' = (a - b)xy$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrive at a separable first-order equation on x . Similarly doing that for other two equations, we will arrive at first order equation on y and z too.

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0401.pdf>

system_of_odes_nonlinear_3eq_order1_type2

`diofant.solvers.ode._nonlinear_3eq_order1_type2(x, y, z, t, eq)`

Equations:

$$ax' = (b - c)yzf(x, y, z, t)$$

$$by' = (c - a)zxf(x, y, z, t)$$

$$cz' = (a - b)xyf(x, y, z, t)$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrive at a first-order differential equation on x . Similarly doing that for other two equations we will arrive at first order equation on y and z .

References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0402.pdf>

Information on the ode module

This module contains `dsolve()` (page 635) and different helper functions that it uses.

`dsolve()` (page 635) 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 635).

Functions in this module

These are the user functions in this module:

- `dsolve()` (page 635) - Solves ODEs.
- `classify_ode()` (page 638) - Classifies ODEs into possible hints for `dsolve()` (page 635).
- `checkodesol()` (page 640) - Checks if an equation is the solution to an ODE.
- `homogeneous_order()` (page 641) - Returns the homogeneous order of an expression.
- `infinitesimals()` (page 641) - Returns the infinitesimals of the Lie group of point transformations of an ODE, such that it is invariant.

These are the non-solver helper functions that are for internal use. The user should use the various options to `dsolve()` (page 635) to obtain the functionality provided by these functions:

- `odesimp()` (page 643) - Does all forms of ODE simplification.
- `ode_sol_simplicity()` (page 646) - A key function for comparing solutions by simplicity.
- `constantsimp()` (page 645) - Simplifies arbitrary constants.
- `constant_renumber()` (page 644) - Renumber arbitrary constants.
- `_handle_Integral()` (page 684) - Evaluate unevaluated Integrals.

See also the docstrings of these functions.

Currently implemented solver methods

The following methods are implemented for solving ordinary differential equations. See the docstrings of the various hint functions for more information on each (run `help(ode)`):

- 1st order separable differential equations.
- 1st order differential equations whose coefficients or dx and dy are functions homogeneous of the same order.
- 1st order exact differential equations.
- 1st order linear differential equations.
- 1st order Bernoulli differential equations.
- Power series solutions for first order differential equations.
- Lie Group method of solving first order differential equations.
- 2nd order Liouville differential equations.
- Power series solutions for second order differential equations at ordinary and regular singular points.
- n th order linear homogeneous differential equation with constant coefficients.
- n th order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.
- n th order linear inhomogeneous differential equation with constant coefficients using the method of variation of parameters.

Philosophy behind this module

This module is designed to make it easy to add new ODE solving methods without having to mess with the solving code for other methods. The idea is that there is a `classify_ode()` (page 638) 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 638) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 405) class. `dsolve()` (page 635), which is the user wrapper function around all of this, will then call `odesimp()` (page 643) 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 635) 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 680) module will become more robust, and unhindered by special case hacks. WolframAlpha and Maple's DETools[odeadvisor] function are two resources you can use to classify a specific ODE. It is also better for a method to work with an n th order ODE instead of only with specific orders, if possible.

To add a new method, there are a few things that you need to do. First, you need a hint name for your method. Try to name your hint so that it is unambiguous with all other methods, including ones that may not be implemented yet. If your method uses integrals, also include a `hint_Integral` hint. If there is more than one way to solve ODEs with your method, include a hint for each one, as well as a `<hint>_best` hint. Your `ode_<hint>_best()` function should choose the best using `min` with `ode_sol_simplicity` as the key argument. See `ode_1st_homogeneous_coeff_best()` (page 648), 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 635) 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 635) code. See also the `classify_ode()` (page 638) 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 643) 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 635). 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 638) (if the match function is more than just a few lines, like `undetermined_coefficients_match()` (page 683), it should go outside of `classify_ode()` (page 638)). It should match the ODE without solving for it as much as possible, so that `classify_ode()` (page 638) 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 638). `classify_ode()` (page 638) will then send this to `dsolve()` (page 635), 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 638) 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 635) 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 Diofant, 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 Diofant documentation. Be sure to make the Sphinx documentation by running `make html` from within the `docs` directory to verify that the docstring formats correctly.

If your solution method involves integrating, use `Integral()` (page 405) instead of `integrate()` (page 399). This allows the user to bypass hard/slow integration by using the `_Integral` variant of your hint. In most cases, calling `diofant.core.basic.Basic.doit()` (page 47) will integrate your solution. If this is not the case, you will need to write special code in `_handle_Integral()` (page 684). 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 643) 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 643) for it. For example, solutions returned from the `1st_homogeneous_coeff` hints often have many `log()` (page 296) terms, so `odesimp()` (page 643) calls `logcombine()` (page 586) 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 645) take better advantage of it. It is better to put simplification in `odesimp()` (page 643) than in your method, because it can then be turned off with the `simplify` flag in `dsolve()` (page 635). 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 Diofant, 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 640) (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 635), that way the test won't be broken simply by the introduction of another matching hint. If your method works for higher order (>1) ODEs, you will need to run `sol = constant_renumber(sol, 'C', 1, order)` for each solution, where `order` is the order of the ODE. This is because `constant_renumber` renumbers the arbitrary constants by printing order, which is platform dependent. Try to test every corner case of your solver, including a range of orders if it is a n th order solver, but if your solver is slow, such as if it involves hard integration, try to keep the test run time down.

Feel free to refactor existing hints to avoid duplicating code or creating inconsistencies. If you can show that your method exactly duplicates an existing method, including in the simplicity and speed of obtaining the solutions, then you can remove the old, less general method. The existing code is tested extensively in `test_ode.py`, so if anything is broken, one of those tests will surely fail.

`diofant.solvers.ode._undetermined_coefficients_match(expr, x)`

Returns a trial function match if undetermined coefficients can be applied to `expr`, and `None` otherwise.

A trial expression can be found for an expression for use with the method of undetermined coefficients if the expression is an additive/multiplicative combination of constants, polynomials in x (the independent variable of `expr`), $\sin(ax + b)$, $\cos(ax + b)$, and e^{ax} terms (in other words, it has a finite number of linearly independent derivatives).

Note that you may still need to multiply each term returned here by sufficient x to make

it linearly independent with the solutions to the homogeneous equation.

This is intended for internal use by `undetermined_coefficients` hints.

Diofant currently has no way to convert $\sin^n(x) \cos^m(y)$ into a sum of only $\sin(ax)$ and $\cos(bx)$ terms, so these are not implemented. 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.

Examples

```
>>> undetermined_coefficients_match(9*x*exp(x) + exp(-x), x)
{'test': True, 'trialset': {E**(-x), E**x, E**x*x}}
>>> undetermined_coefficients_match(log(x), x)
{'test': False}
```

`diofant.solvers.ode._handle_Integral(expr, func, order, hint)`

Converts a solution with Integrals in it into an actual solution.

For most hints, this simply runs `expr.doit()`.

4.16.5 Recurrence Equations

This module is intended for solving recurrences (difference equations).

`diofant.solvers.recurr.rsolve(f, *y, init={}, simplify=True)`

Solve recurrence equations.

The equations can involve objects of the form $y(n + k)$, where k is a constant.

Parameters

- **f** (*Expr, Equality or iterable of above*) - The single recurrence equation or a system of recurrence equations.
- ***y** (*tuple*) - Holds function applications $y(n)$, wrt to which the recurrence equation(s) will be solved. If none given (empty tuple), this will be guessed from the provided equation(s).
- **init** (*dict, optional*) - The initial/boundary conditions for the recurrence equations as mapping of the function application $y(n_i)$ to its value. Default is empty dictionary.
- **simplify** (*bool, optional*) - Enable simplification (default) on solutions.

Examples

```
>>> eq = (n - 1)*f(n + 2) - (n**2 + 3*n - 2)*f(n + 1) + 2*n*(n + 1)*f(n)
```

```
>>> rsolve(eq)
[{f: Lambda(n, 2**n*C0 + C1*factorial(n))}]
>>> rsolve(eq, init={f(0): 0, f(1): 3})
[{f: Lambda(n, 3*2**n - 3*factorial(n))}]
```


Notes

Currently, the function can handle linear recurrences with polynomial coefficients and hypergeometric inhomogeneous part.

See also:

`diofant.solvers.ode.dsolve` (page 635)
solving differential equations

`diofant.solvers.solvers.solve` (page 606)
solving algebraic equations

`diofant.solvers.recurr.rsolve_hyper(coeffs, f, n)`

Find hypergeometric solutions for linear recurrence.

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.

Notes

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 .

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.

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.

Examples

```
>>> rsolve_hyper([-1, 1], 1 + n, n)
(C0 + n*(n + 1)/2, [C0])
```

References

- [\[Petkovsek92\]](#)
- [\[PetkovsekWZ97\]](#)

`diofant.solvers.recurr.rsolve_poly(coeffs, f, n)`

Find polynomial solutions for linear recurrence.

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.

Notes

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 Petkovšek.

It is possible to generalize the algorithm implemented here to the case of linear q -difference and differential equations.

Examples

Lets say that we would like to compute m -th Bernoulli polynomial up to a constant, using $b(n+1) - b(n) = mn^{m-1}$ recurrence:

```
>>> rsolve_poly([-1, 1], 4*n**3, n)
(C0 + n**4 - 2*n**3 + n**2, [C0])
>>> bernoulli(4, n)
n**4 - 2*n**3 + n**2 - 1/30
```

References

- [\[ABPetkovsek95\]](#)
- [\[Petkovsek92\]](#)
- [\[PetkovsekWZ97\]](#)

`diofant.solvers.recurr.resolve_ratio(coeffs, f, n)`

Find rational solutions for linear recurrence.

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.

Notes

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 the Hyper algorithm were it is used to find particular solution of inhomogeneous part of a recurrence.

Examples

```
>>> resolve_ratio([-2*n**3 + n**2 + 2*n - 1, 2*n**3 + n**2 - 6*n,
...               -2*n**3 - 11*n**2 - 18*n - 9,
...               2*n**3 + 13*n**2 + 22*n + 8], 0, n)
(C2*(2*n - 3)/(2*(n**2 - 1)), [C2])
```

References

- [Abr95]

See also:

[`resolve_hyper`](#) (page 685)

4.16.6 PDE

User Functions

These are functions that are imported into the global namespace with `from diofant import *`. They are intended for user use.

pde_separate

`diofant.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.

Examples

```
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(Derivative(u(x, t), x), E**(u(x, t))*Derivative(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[E**(-X(x))*Derivative(X(x), x), E**T(t)*Derivative(T(t), t)]
```

```
>>> eq = Eq(Derivative(u(x, t), (x, 2)), Derivative(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)]
```

See also:

[`diofant.solvers.pde.pde_separate_add`](#) (page 688), [`diofant.solvers.pde.pde_separate_mul`](#) (page 689)

pde_separate_add

`diofant.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

```
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(Derivative(u(x, t), x), E**(u(x, t))*Derivative(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[E**(-X(x))*Derivative(X(x), x), E**T(t)*Derivative(T(t), t)]
```

pde_separate_mul

`diofant.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

```
>>> u, X, Y = map(Function, 'uXY')
```

```
>>> eq = Eq(Derivative(u(x, y), (x, 2)), Derivative(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

`diofant.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:

```
>>> 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

```
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> pdsolve(eq)
Eq(f(x, y), E**(-2*x/13 - 3*y/13)*F(3*x - 2*y))
```

classify_pde

`diofant.solvers.pde.classify_pde(eq, func=None, dict=False, **kwargs)`

Returns a tuple of possible `pdsolve()` classifications for a PDE.

The tuple is ordered so that first item is the classification that `pdsolve()` uses to solve the PDE by default. In general, classifications near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make `pdsolve` use a different classification, use `pdsolve(PDE, func, hint=<classification>)`. See also the `pdsolve()` docstring for different meta-hints you can use.

If `dict` is true, `classify_pde()` will return a dictionary of `hint:match` expression terms. This is intended for internal use by `pdsolve()`. Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by doing `help(pde.pde_hintname)`, where `hintname` is the name of the hint without “_Integral”.

See `diofant.pde.allhints` or the `diofant.pde` docstring for a list of all supported hints that can be returned from `classify_pde`.

Examples

```

>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)

```

checkpdesol

`diofant.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

```
>>> 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, E**(-6*x/25 - 8*y/25)*(x*F(4*x - 3*y) - 6*F(4*x - 3*y)/25 +
↳ 4*Subs(Derivative(F(_xi_1), _xi_1), (_xi_1, 4*x - 3*y))))
```

Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

pde_1st_linear_constant_coeff_homogeneous

`diofant.solvers.pde.pde_1st_linear_constant_coeff_homogeneous`(*eq, func, order, match, solvefun*)

Solves a first order linear homogeneous partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{df(x, y)}{dx} + b \frac{df(x, y)}{dy} + cf(x, y) = 0$$

where a , b and c are constants.

The general solution is of the form:

```
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform)
a· $\frac{\partial}{\partial x}$ (f(x, y)) + b· $\frac{\partial}{\partial y}$ (f(x, y)) + c·f(x, y)
>>> pprint(pdsolve(genform))

$$f(x, y) = e^{\frac{-c \cdot (a \cdot x + b \cdot y)}{a^2 + b^2}} \cdot F(-a \cdot y + b \cdot x)$$

```

Examples

```
>>> pdsolve(f(x, y) + f(x, y).diff(x) + f(x, y).diff(y))
Eq(f(x, y), E**(-x/2 - y/2)*F(x - y))
>>> pprint(pdsolve(f(x, y) + f(x, y).diff(x) + f(x, y).diff(y)))

$$f(x, y) = e^{-\frac{x}{2} - \frac{y}{2}} \cdot F(x - y)$$

```


References

- Viktor Grigoryan, "Partial Differential Equations" Math 124A - Fall 2010, pp.7

pde_1st_linear_constant_coeff

`diofant.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

$$a \frac{df(x, y)}{dx} + b \frac{df(x, y)}{dy} + cf(x, y) = G(x, y)$$

where a , b and c are constants and $G(x, y)$ can be an arbitrary function in x and y .

The general solution of the PDE is:

```
>>> 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)
a·f(x, y) + b· $\frac{\partial}{\partial x}$ (f(x, y)) + c· $\frac{\partial}{\partial y}$ (f(x, y)) - G(x, y)
>>> pprint(pdsolve(genform, hint='1st_linear_constant_coeff_Integral'))
```

$$f(x, y) = e^{\frac{-a \cdot \xi}{b^2 + c^2}} \cdot \left(F(\eta) + \int \frac{a \cdot \xi}{b^2 + c^2} \cdot G\left(\frac{b \cdot \xi + c \cdot \eta}{b^2 + c^2}, \frac{-b \cdot \eta + c \cdot \xi}{b^2 + c^2}\right) d\xi \right) \Big|_{\eta = -b \cdot y + c \cdot x, \xi = b \cdot x + c \cdot y}$$

Examples

```
>>> eq = -2*f(x, y).diff(x) + 4*f(x, y).diff(y) + 5*f(x, y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), E**(x/2 - y)*(E**(x/2 + 4*y)/15 + F(4*x + 2*y)))
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

pde_1st_linear_variable_coeff

`diofant.solvers.pde.pde_1st_linear_variable_coeff(eq, func, order, match, solvefun)`

Solves a first order linear partial differential equation with variable coefficients. The general form of this partial differential equation is

$$a(x, y) \frac{df(x, y)}{dx} + b(x, y) \frac{df(x, y)}{dy} + c(x, y)f(x, y) - G(x, y)$$

where $a(x, y)$, $b(x, y)$, $c(x, y)$ and $G(x, y)$ are arbitrary functions in x and y . This PDE is converted into an ODE by making the following transformation.

1] ξ as x

2] η as the constant in the solution to the differential equation $\frac{dy}{dx} = -\frac{b}{a}$

Making the following substitutions reduces it to the linear ODE

$$a(\xi, \eta) \frac{du}{d\xi} + c(\xi, \eta)u - d(\xi, \eta) = 0$$

which can be solved using `dsolve`.

The general form of this PDE is:

```
>>> a, b, c, G, f = [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x, y)
>>> pprint(genform)
-G(x, y) + a(x, y)·f(x, y) + b(x, y)· $\frac{\partial}{\partial x}(f(x, y))$  + c(x, y)· $\frac{\partial}{\partial y}(f(x, y))$ 
```

Examples

```
>>> f = Function('f')
>>> eq = x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), E**(y**2/2)*F(x*y) + 1)
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

Information on the pde module

This module contains `pdsolve()` and different helper functions that it uses. It is heavily inspired by the `ode` module and hence the basic infrastructure remains the same.

Functions in this module

These are the user functions in this module:

- `pdsolve()` - Solves PDE's
- `classify_pde()` - Classifies PDEs into possible hints for `dsolve()`.
- **`pde_separate()` - Separate variables in partial differential equation either by additive or multiplicative separation approach.**

These are the helper functions in this module:

- `pde_separate_add()` - Helper function for searching additive separable solutions.
- **`pde_separate_mul()` - Helper function for searching multiplicative separable solutions.**

Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various `pde_hint()` functions for more information on each (run `help(pde)`):

- 1st order linear homogeneous partial differential equations with constant coefficients.
- 1st order linear general partial differential equations with constant coefficients.
- 1st order linear partial differential equations with variable coefficients.

4.16.7 Utilities for solving

General utility functions for solvers.

`diofant.solvers.utils.checksol(f, sol, **flags)`

Checks whether `sol` is a solution of equations `f`.

Examples

```
>>> 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
```

Returns

bool or None - Return `True`, if solution satisfy all equations in `f`. Return `False`, if a solution doesn't satisfy any equation. Else (i.e. one or more checks are inconclusive), return `None`.

Parameters

- **`f`** (*Expr or iterable of Expr's*) - Equations to substitute solutions in.

- **sol** (*dict of Expr's*) – Mapping of symbols to values.
- ****flags** (*dict*) – A dictionary of following parameters:
 - minimal**
[bool, optional] Do a very fast, minimal testing. Default is False.
 - warn**
[bool, optional] Show a warning if it could not conclude. Default is False.
 - simplify**
[bool, optional] Simplify solution before substituting into function and simplify the function before trying specific simplifications. Default is True.
 - force**
[bool, optional] Make positive all symbols without assumptions regarding sign. Default is False.

Utility functions for classifying and solving ordinary and partial differential equations.

Contains

`_preprocess ode_order _desolve`

`diofant.solvers.deutils.ode_order(expr, func)`

Returns the order of a given differential equation with respect to func.

This function is implemented recursively.

Examples

```
>>> 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
```

4.17 Tensors

A module to manipulate symbolic objects with indices including tensors.

4.17.1 N-dim array

N-dim array module.

Four classes are provided to handle N-dim arrays, given by the combinations dense/sparse (i.e. whether to store all elements or only the non-zero ones in memory) and mutable/immutable (immutable classes are Diofant objects, but cannot change after they have been created).

Examples

The following examples show the usage of Array. This is an abbreviation for ImmutableDenseNDimArray, that is an immutable and dense N-dim array, the other classes are analogous. For mutable classes it is also possible to change element values after the object has been constructed.

Array construction can detect the shape of nested lists and tuples:

```
>>> a1 = Array([[1, 2], [3, 4], [5, 6]])
>>> a1
[[1, 2], [3, 4], [5, 6]]
>>> a1.shape
(3, 2)
>>> a1.rank()
2
>>> a2 = Array([[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]])
>>> a2
[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]
>>> a2.shape
(2, 2, 2)
>>> a2.rank()
3
```

Otherwise one could pass a 1-dim array followed by a shape tuple:

```
>>> m1 = Array(range(12), (3, 4))
>>> m1
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
>>> m2 = Array(range(12), (3, 2, 2))
>>> m2
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> m2[1, 1, 1]
7
>>> m2.reshape(4, 3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

Slice support:

```
>>> m2[:, 1, 1]
[3, 7, 11]
```

Elementwise derivative:

```
>>> m3 = Array([x**3, x*y, z])
>>> m3.diff(x)
[3*x**2, y, 0]
>>> m3.diff(z)
[0, 0, 1]
```

Multiplication with other Diofant expressions is applied elementwisely:

```
>>> (1+x)*m3
[x**3*(x + 1), x*y*(x + 1), z*(x + 1)]
```

To apply a function to each element of the N-dim array, use `applyfunc`:

```
>>> m3.applyfunc(lambda x: x/2)
[[x**3/2, x*y/2, z/2]
```

N-dim arrays can be converted to nested lists by the `tolist()` method:

```
>>> m2.tolist()
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
```

If the rank is 2, it is possible to convert them to matrices with `tomatrix()`:

```
>>> m1.tomatrix()
Matrix([
[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]])
```

Products and contractions

Tensor product between arrays A_{i_1, \dots, i_n} and B_{j_1, \dots, j_m} creates the combined array $P = A \otimes B$ defined as

$$P_{i_1, \dots, i_n, j_1, \dots, j_m} := A_{i_1, \dots, i_n} \cdot B_{j_1, \dots, j_m}$$

It is available through `tensorproduct(...)`:

```
>>> A = Array([x, y, z, t])
>>> B = Array([1, 2, 3, 4])
>>> tensorproduct(A, B)
[[x, 2*x, 3*x, 4*x], [y, 2*y, 3*y, 4*y], [z, 2*z, 3*z, 4*z],
 [t, 2*t, 3*t, 4*t]]
```

Tensor product between a rank-1 array and a matrix creates a rank-3 array:

```
>>> p1 = tensorproduct(A, eye(4))
>>> p1
[[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]],
 [[y, 0, 0, 0], [0, y, 0, 0], [0, 0, y, 0], [0, 0, 0, y]],
 [[z, 0, 0, 0], [0, z, 0, 0], [0, 0, z, 0], [0, 0, 0, z]],
 [[t, 0, 0, 0], [0, t, 0, 0], [0, 0, t, 0], [0, 0, 0, t]]]
```

Now, to get back $A_0 \otimes \mathbf{1}$ one can access $p_{0,m,n}$ by slicing:

```
>>> p1[0, :, :]
[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]]
```

Tensor contraction sums over the specified axes, for example contracting positions a and b means

$$A_{i_1, \dots, i_a, \dots, i_b, \dots, i_n} \Rightarrow \sum_k A_{i_1, \dots, k, \dots, k, \dots, i_n}$$

Remember that Python indexing is zero starting, to contract the a -th and b -th axes it is therefore necessary to specify $a - 1$ and $b - 1$

```
>>> C = Array([[x, y], [z, t]])
```

The matrix trace is equivalent to the contraction of a rank-2 array:

$$A_{m,n} \Rightarrow \sum_k A_{k,k}$$

```
>>> tensorcontraction(C, (0, 1))
t + x
```

Matrix product is equivalent to a tensor product of two rank-2 arrays, followed by a contraction of the 2nd and 3rd axes (in Python indexing axes number 1, 2).

$$A_{m,n} \cdot B_{i,j} \implies \sum_k A_{m,k} \cdot B_{k,j}$$

```
>>> D = Array([[2, 1], [0, -1]])
>>> tensorcontraction(tensorproduct(C, D), (1, 2))
[[2*x, x - y], [2*z, -t + z]]
```

One may verify that the matrix product is equivalent:

```
>>> Matrix([x, y], [z, t])*Matrix([[2, 1], [0, -1]])
Matrix([
[2*x, x - y],
[2*z, -t + z]])
```

or equivalently

```
>>> C.tomatrix()*D.tomatrix()
Matrix([
[2*x, x - y],
[2*z, -t + z]])
```

Derivatives by array

The usual derivative operation may be extended to support derivation with respect to arrays, provided that all elements in the that array are symbols or expressions suitable for derivations.

The definition of a derivative by an array is as follows: given the array A_{i_1, \dots, i_N} and the array X_{j_1, \dots, j_M} the derivative of arrays will return a new array B defined by

$$B_{j_1, \dots, j_M, i_1, \dots, i_N} := \frac{\partial A_{i_1, \dots, i_N}}{\partial X_{j_1, \dots, j_M}}$$

The function `derive_by_array` performs such an operation. With scalars, it behaves exactly as the ordinary derivative:

```
>>> derive_by_array(sin(x*y), x)
y*cos(x*y)
```

Scalar derived by an array basis:

```
>>> derive_by_array(sin(x*y), [x, y, z])
[y*cos(x*y), x*cos(x*y), 0]
```

Deriving array by an array basis: $B^{nm} := \frac{\partial A^m}{\partial x^n}$

```
>>> basis = [x, y, z]
>>> ax = derive_by_array([exp(x), sin(y*z), t], basis)
>>> ax
[[E**x, 0, 0], [0, z*cos(y*z), 0], [0, y*cos(y*z), 0]]
```

Contraction of the resulting array: $\sum_m \frac{\partial A^m}{\partial x^m}$

```
>>> tensorcontraction(ax, (0, 1))
E**x + z*cos(y*z)
```

Classes

class diofant.tensor.array.Array

alias of [ImmutableDenseNDimArray](#) (page 700)

class diofant.tensor.array.ImmutableDenseNDimArray(*iterable=None, shape=None, **kwargs*)

An immutable version of a dense N-dim array.

class diofant.tensor.array.ImmutableSparseNDimArray(*iterable=None, shape=None, **kwargs*)

An immutable version of a sparse N-dim array.

class diofant.tensor.array.MutableDenseNDimArray(*iterable=None, shape=None, **kwargs*)

A mutable version of a dense N-dim array.

class diofant.tensor.array.MutableSparseNDimArray(*iterable=None, shape=None, **kwargs*)

A mutable version of a sparse N-dim array.

Functions

diofant.tensor.array.derive_by_array(*expr, dx*)

Derivative by arrays. Supports both arrays and scalars.

Given the array A_{i_1, \dots, i_N} and the array X_{j_1, \dots, j_M} this function will return a new array B defined by

$$B_{j_1, \dots, j_M, i_1, \dots, i_N} := \frac{\partial A_{i_1, \dots, i_N}}{\partial X_{j_1, \dots, j_M}}$$

Examples

```
>>> derive_by_array(cos(x*t), x)
-t*sin(t*x)
>>> derive_by_array(cos(x*t), [x, y, z, t])
[-t*sin(t*x), 0, 0, -x*sin(t*x)]
>>> derive_by_array([x, y**2*z], [[x, y], [z, t]])
[[[1, 0], [0, 2*y*z]], [[0, y**2], [0, 0]]]
```

diofant.tensor.array.permutedims(*expr, perm*)

Permutes the indices of an array.

Parameter specifies the permutation of the indices.

Examples

```
>>> a = Array([[x, y, z], [t, sin(x), 0]])
>>> a
[[x, y, z], [t, sin(x), 0]]
>>> permutedims(a, (1, 0))
[[x, t], [y, sin(x)], [z, 0]]
```

If the array is of second order, transpose can be used:

```
>>> transpose(a)
[[x, t], [y, sin(x)], [z, 0]]
```

Examples on higher dimensions:

```
>>> b = Array([[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]])
>>> permutedims(b, (2, 1, 0))
[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
>>> permutedims(b, (1, 2, 0))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

Permutation objects are also allowed:

```
>>> permutedims(b, Permutation([1, 2, 0]))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

`diofant.tensor.array.tensorcontraction(array, *contraction_axes)`

Contraction of an array-like object on the specified axes.

Examples

```
>>> tensorcontraction(eye(3), (0, 1))
3
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]],
 [[12, 13, 14], [15, 16, 17]]]
>>> tensorcontraction(A, (0, 2))
[21, 30]
```

Matrix multiplication may be emulated with a proper combination of `tensorcontraction` and `tensorproduct`

```
>>> from diofant.abc import e, f, g, h
>>> m1 = Matrix([[a, b], [c, d]])
>>> m2 = Matrix([[e, f], [g, h]])
>>> p = tensorproduct(m1, m2)
>>> p
[[[a*e, a*f], [a*g, a*h]], [[b*e, b*f], [b*g, b*h]]],
 [[c*e, c*f], [c*g, c*h]], [[d*e, d*f], [d*g, d*h]]]]
>>> tensorcontraction(p, (1, 2))
[[a*e + b*g, a*f + b*h], [c*e + d*g, c*f + d*h]]
>>> m1*m2
Matrix([
[a*e + b*g, a*f + b*h],
[c*e + d*g, c*f + d*h]])
```

`diofant.tensor.array.tensorproduct(*args)`

Tensor product among scalars or array-like objects.

Examples

```
>>> A = Array([[1, 2], [3, 4]])
>>> B = Array([x, y])
>>> tensorproduct(A, B)
[[[x, y], [2*x, 2*y]], [[3*x, 3*y], [4*x, 4*y]]]
>>> tensorproduct(A, x)
[[x, 2*x], [3*x, 4*x]]
>>> tensorproduct(A, B, B)
[[[[x**2, x*y], [x*y, y**2]], [[2*x**2, 2*x*y], [2*x*y, 2*y**2]]],
 [[3*x**2, 3*x*y], [3*x*y, 3*y**2]], [[4*x**2, 4*x*y], [4*x*y, 4*y**2]]]]
```

Applying this function on two matrices will result in a rank 4 array.

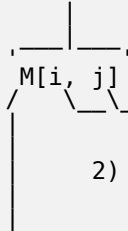
```
>>> m = Matrix([[x, y], [z, t]])
>>> p = tensorproduct(eye(3), m)
>>> p
[[[[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]],
 [[0, 0], [0, 0]], [[x, y], [z, t]], [[0, 0], [0, 0]]],
 [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[x, y], [z, t]]]]
```

4.17.2 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.



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:

```
>>> 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]
x[j]*M[i, 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:

```
>>> 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()`:

```
>>> get_indices(A[i, j, j])
({i}, {j})
>>> get_contraction_structure(A[i, j, j])
{(j,): {A[i, j, j]}}
```

See the appropriate docstrings for a detailed explanation of the output.

class diofant.tensor.indexed.Idx(*label*, *range*=None, ***kw_args*)

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

```
>>> i, L, U = symbols('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
```

property label

Returns the label (Integer or integer expression) of the `Idx` object.

Examples

```
>>> Idx(2).label
2
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

property lower

Returns the lower bound of the Index.

Examples

```
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

property upper

Returns the upper bound of the Index.

Examples

```
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

exception diofant.tensor.indexed.IndexExceptionError

Generic index error.

class diofant.tensor.indexed.Indexed(base, *args, **kw_args)

Represents a mathematical object with indices.

```
>>> 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
```

property base

Returns the IndexedBase of the Indexed object.

Examples

```
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

property indices

Returns the indices of the Indexed object.

Examples

```
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```

property 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

```
>>> 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)]
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

property rank

Returns the rank of the Indexed object.

Examples

```
>>> 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
```

property 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.

```
>>> 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 diofant.tensor.indexed.IndexedBase(label, shape=None, **kw_args)

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.

```
>>> A = IndexedBase('A')
>>> A
A
>>> type(A)
<class 'diofant.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 'diofant.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!)

```
>>> o, p = symbols('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)
```

property args

Returns the arguments used to create this IndexedBase object.

Examples

```
>>> IndexedBase('A', shape=(x, y)).args
(A, (x, y))
```

property label

Returns the label of the IndexedBase object.

Examples

```
>>> IndexedBase('A', shape=(x, y)).label
A
```

property shape

Returns the shape of the IndexedBase object.

Examples

```
>>> 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)
```

4.17.3 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.

exception diofant.tensor.index_methods.IndexConformanceExceptionError

Raised if indexes are not consistent.

diofant.tensor.index_methods.get_contraction_structure(expr)

Determine dummy indices of expr and describe its structure

By *dummy* we mean indices that are summation indices.

The structure of the expression is determined and described as follows:

- 1) A conforming summation of Indexed objects is described with a dict where the keys are summation indices and the corresponding values are sets containing all terms for which the summation applies. All Add objects in the Diofant expression tree are described like this.
- 2) For all nodes in the Diofant expression tree that are *not* of type Add, the following applies:

If a node discovers contractions in one of its arguments, the node itself will be stored as a key in the dict. For that key, the corresponding value is a list of dicts, each of which is the result of a recursive call to get_contraction_structure(). The list contains only dicts for the non-trivial deeper contractions, omitting dicts with None as the one and only key.

Note: The presence of expressions among the dictionary keys indicates multiple levels of index contractions. A nested dict displays nested contractions and may itself contain dicts from a deeper level. In practical calculations the summation in the deepest nested level must be calculated first so that the outer expression can access the resulting indexed object.

Examples

```
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j = map(Idx, ['i', 'j'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j])
{(i,): {x[i]*y[i]}, (j,): {A[j, j]}}
>>> get_contraction_structure(x[i]*y[j])
{None: {x[i]*y[j]}}
```

A multiplication of contracted factors results in nested dicts representing the internal contractions.

```
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d, key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```
>>> d[None]
{x[i, i]*y[j, j]}
```

Factors are contracted “first”:

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{(i,): {x[i, i]}}, {(j,): {y[j, j]}}]
```

A parenthesized Add object is also returned as a nested dictionary. The term containing the parenthesis is a Mul with a contraction among the arguments, so it will be found as a key in the result. It stores the dictionary resulting from a recursive call on the Add expression.

```
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d, key=default_sort_key)
[(x[j]*A[i, j] + y[i])*x[i], (i,)]
>>> d[(i,)]
{(x[j]*A[i, j] + y[i])*x[i]}
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[None: {y[i]}, (j,): {x[j]*A[i, j]}]
```

Powers with contractions in either base or exponent will also be found as keys in the dictionary, mapping to a list of results from recursive calls:

```
>>> d = get_contraction_structure(A[j, j]**A[i, i])
>>> d[None]
{A[j, j]**A[i, i]}
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): {A[j, j]}}
>>> nested_contractions[1]
{(i,): {A[i, i]}}
```

The description of the contraction structure may appear complicated when represented with a string in the above examples, but it is easy to iterate over:

```
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...         # treat outermost contractions here
```

`diofant.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

```
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j = symbols('i j', 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 `IndexConformanceExceptionError` is raised.

```
>>> get_indices(x[i] + A[i, j]*y[j])
({i}, {})
```

Exceptions

An `IndexConformanceExceptionError` means that the terms are not compatible, e.g.

```
>>> get_indices(x[i] + y[j])
Traceback (most recent call last):
IndexConformanceExceptionError: Indices are not consistent: x(i) + y(j)
```

Warning: The concept of *outer* indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```
>>> get_indices((x[i] + A[i, j]*y[j])*x[j])
({i, j}, {})
```

This is correct and may appear convenient, but you need to be careful with this as Diofant 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.

4.18 Utilities

This module contains some general purpose utilities that are used across Diofant.

4.18.1 Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the *Tensors* (page 696). Here is a simple example that shows how to setup a binary routine that calculates a matrix-vector product.

```
>>> from diofant.utilities.autowrap import autowrap
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j])
>>> instruction
Eq(y[i], x[j]*A[i, 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 Diofant 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.

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.

```
>>> 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 Diofant object. If it is desired to use the compiled function in symbolic expressions, it is better to use binary_function() which returns a Diofant Function object. The binary callable is attached as the `_imp` attribute and invoked when a numerical evaluation is requested with evalf(), or with lambdify().

```
>>> 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}, strict=False)
0.e-190
```

The idea is that a Diofant 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 Diofant'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 Diofant.
- 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.

exception `diofant.utilities.autowrap.CodeWrapError`

Generic code wrapping error.

class `diofant.utilities.autowrap.CodeWrapper(generator, filepath=None, flags=[], verbose=False)`

Base Class for code wrappers.

class `diofant.utilities.autowrap.CythonCodeWrapper(*args, **kwargs)`

Wrapper that uses Cython.

dump_pyx(routines, f, prefix)

Write a Cython file with python wrappers

This file contains all the definitions of the routines in c code and refers to the header file.

Parameters

- **routines** (*list*) - List of Routine instances
- **f** (*file*) - File-like object to write the file to
- **prefix** (*str*) - The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

class `diofant.utilities.autowrap.DummyWrapper(generator, filepath=None, flags=[], verbose=False)`

Class used for testing independent of backends.

```
class diofant.utilities.autowrap.F2PyCodeWrapper(generator, filepath=None,
                                                flags=[], verbose=False)
```

Wrapper that uses f2py.

```
class diofant.utilities.autowrap.UfuncifyCodeWrapper(generator, filepath=None,
                                                    flags=[], verbose=False)
```

Wrapper for Ufuncify.

```
dump_c(routines, f, prefix)
```

Write a C file with python wrappers

This file contains all the definitions of the routines in c code.

Parameters

- **routines** (*list*) – List of Routine instances
- **f** (*file*) – File-like object to write the file to
- **prefix** (*str*) – The filename prefix, used to name the imported module.

```
diofant.utilities.autowrap.autowrap(expr, language=None, backend='f2py',
                                     tempdir=None, args=None, flags=None,
                                     verbose=False, helpers=[])
```

Generates python callable binaries based on the math expression.

Parameters

- **expr** – The Diofant expression that should be wrapped as a binary routine.
- **language** (*string, optional*) – If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.
- **backend** (*string, optional*) – Backend used to wrap the generated code. Either 'f2py' [default], or 'cython'.
- **tempdir** (*string, optional*) – Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.
- **args** (*iterable, optional*) – An ordered iterable of symbols. Specifies the argument sequence for the function.
- **flags** (*iterable, optional*) – Additional option flags that will be passed to the backend.
- **verbose** (*bool, optional*) – If True, autowrap will not mute the command line backends. This can be helpful for debugging.
- **helpers** (*iterable, optional*) – Used to define auxillary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in the helpers iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<function_name>, <diofant_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

Examples

```
>>> expr = ((x - y + z)**13).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

`diofant.utilities.autowrap.binary_function(symfunc, expr, **kwargs)`

Returns a diofant function with `expr` as binary implementation

This is a convenience function that automates the steps needed to autowrap the Diofant expression and attaching it to a Function object with `implemented_function()`.

```
>>> expr = ((x - y)**25).expand()
>>> f = binary_function('f', expr)
>>> type(f)
<class 'diofant.core.function.UndefinedFunction'>
>>> 2*f(x, y)
2*f(x, y)
>>> f(x, y).evalf(2, subs={x: 1, y: 2})
-1.0
```

`diofant.utilities.autowrap.ufuncify(args, expr, language=None, backend='numpy',
tempdir=None, flags=None, verbose=False,
helpers=[])`

Generates a binary function that supports broadcasting on numpy arrays.

Parameters

- **args** (*iterable*) – Either a Symbol or an iterable of symbols. Specifies the argument sequence for the function.
- **expr** – A Diofant expression that defines the element wise operation.
- **language** (*string, optional*) – If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.
- **backend** (*string, optional*) – Backend used to wrap the generated code. Either 'numpy' [default], 'cython', or 'f2py'.
- **tempdir** (*string, optional*) – Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.
- **flags** (*iterable, optional*) – Additional option flags that will be passed to the backend
- **verbose** (*bool, optional*) – If True, autowrap will not mute the command line backends. This can be helpful for debugging.
- **helpers** (*iterable, optional*) – Used to define auxillary expressions needed for the main `expr`. If the main expression needs to call a specialized function it should be put in the `helpers` iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (`<function_name>`, `<diofant_expression>`, `<arguments>`). It is mandatory to supply an argument sequence to helper routines.

Notes

The default backend ('numpy') will create actual instances of `numpy.ufunc`. These support ndimensional broadcasting, and implicit type conversion. Use of the other backends will result in a "ufunc-like" function, which requires equal length 1-dimensional arrays for all arguments, and will not perform any type conversions.

References

- <https://docs.scipy.org/doc/numpy/reference/ufuncs.html>

Examples

```
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f) is np.ufunc
True
>>> f([1, 2, 3], 2)
[ 3.  6. 11.]
>>> f(np.arange(5), 3)
[ 3.  4.  7. 12. 19.]
```

For the F2Py and Cython backends, inputs are required to be equal length 1-dimensional arrays. The F2Py backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```
>>> f_fortran = ufuncify((x, y), y + x**2, backend='F2Py')
>>> f_fortran(1, 2)
[ 3.]
>>> f_fortran(np.array([1, 2, 3]), np.array([1.0, 2.0, 3.0]))
[ 2.  6. 12.]
```

4.18.2 Codegen

This module provides functionality to generate directly compilable code from Diofant expressions. The `codegen` function is the user interface to the code generation functionality in Diofant. 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 `diofant` namespace automatically, to use it you must first execute

```
>>> from diofant.utilities.codegen import codegen # noqa: F401
```

Implementation Details

Here we present the most important pieces of the internal structure, as advanced users may want to use it directly, for instance by subclassing a code generator for a specialized application. **It is very likely that you would prefer to use the `codegen()` function documented above.**

Basic assumptions:

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

Routine

The Routine class is a very important piece of the codegen module. Viewing the codegen utility as a translator of mathematical expressions into a set of statements in a programming language, the Routine instances are responsible for extracting and storing information about how the math can be encapsulated in a function call. Thus, it is the Routine constructor that decides what arguments the routine will need and if there should be a return value.

API Reference

module for generating C, C++, Fortran77, Fortran90 and Octave/Matlab routines that evaluate diofant expressions. This module is work in progress. Only the milestones with a '+' character in the list below have been completed.

— How is `diofant.utilities.codegen` different from `diofant.printing.ccode`? —

We considered the idea to extend the printing routines for diofant 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.
- Diofant 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 diofant functions in one C routine, eventually sharing common intermediate results with the help of the cse routine. This is more than just printing.
- From the programming perspective, expressions with constants should be evaluated in the code generator as much as possible. This is different for printing.

— Basic assumptions —

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

— Milestones —

- First working version with scalar input arguments, generating C code, tests
- Friendly functions that are easier to use than the rigorous Routine/CodeGen workflow.
- Integer and Real numbers as input and output
- Output arguments
- InputOutput arguments
- Sort input/output arguments properly
- Contiguous array arguments (numpy matrices)
- Also generate .pyf code for f2py (in autowrap module)
- Isolate constants and evaluate them beforehand in double precision
- Fortran 90
- Octave/Matlab
- Common Subexpression Elimination
- User defined comments in the generated code
- Optional extra include lines for libraries/objects that can eval special functions
- Test other C compilers and libraries: gcc, tcc, libtcc, gcc+gsl, ...
- Contiguous array arguments (diofant matrices)
- Non-contiguous array arguments (diofant 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 `diofant.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 diofant.utilities.codegen.CCodeGen(project='project', printer=None,  
                                         preprocessor_statements=None,  
                                         cse=False)
```

Generator for C code.

The `.write()` method inherited from `CodeGen` will output a code file and an interface file, `<prefix>.c` and `<prefix>.h` respectively.

```
dump_c(routines, f, prefix, header=True, empty=True)
```

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

- **routines** (*list*) - A list of Routine instances.
- **f** (*file-like*) - Where to write the file.
- **prefix** (*string*) - The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.
- **header** (*bool, optional*) - When True, a header comment is included on top of each source file. [default : True]
- **empty** (*bool, optional*) - When True, empty lines are included to structure the source files. [default : True]

```
dump_h(routines, f, prefix, header=True, empty=True)
```

Writes the C header file.

This file contains all the function declarations.

Parameters

- **routines** (*list*) - A list of Routine instances.
- **f** (*file-like*) - Where to write the file.
- **prefix** (*string*) - The filename prefix, used to construct the include guards. Only the basename of the prefix is used.
- **header** (*bool, optional*) - When True, a header comment is included on top of each source file. [default : True]
- **empty** (*bool, optional*) - When True, empty lines are included to structure the source files. [default : True]

```
get_prototype(routine)
```

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an `CodeGenError` is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

```
class diofant.utilities.codegen.CodeGen(project='project', cse=False)
```

Abstract class for the code generators.

```
dump_code(routines, f, prefix, header=True, empty=True)
```

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

- **routines** (*list*) – A list of Routine instances.
- **f** (*file-like*) – Where to write the file.
- **prefix** (*string*) – The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.
- **header** (*bool, optional*) – When True, a header comment is included on top of each source file. [default : True]
- **empty** (*bool, optional*) – When True, empty lines are included to structure the source files. [default : True]

routine(*name, expr, argument_sequence, global_vars=None*)

Creates an Routine object that is appropriate for this language.

This implementation is appropriate for at least C/Fortran. Subclasses can override this if necessary.

Here, we assume at most one return value (the l-value) which must be scalar. Additional outputs are OutputArguments (e.g., pointers on right-hand-side or pass-by-reference). Matrices are always returned via OutputArguments. If *argument_sequence* is None, arguments will be ordered alphabetically, but with all InputArguments first, and then OutputArgument and InOutArguments.

write(*routines, prefix, to_files=False, header=True, empty=True*)

Writes all the source code files for the given routines.

The generated source is returned as a list of (filename, contents) tuples, or is written to files (see below). Each filename consists of the given prefix, appended with an appropriate extension.

Parameters

- **routines** (*list*) – A list of Routine instances to be written
- **prefix** (*string*) – The prefix for the output files
- **to_files** (*bool, optional*) – When True, the output is written to files. Otherwise, a list of (filename, contents) tuples is returned. [default: False]
- **header** (*bool, optional*) – When True, a header comment is included on top of each source file. [default: True]
- **empty** (*bool, optional*) – When True, empty lines are included to structure the source files. [default: True]

class diofant.utilities.codegen.**DataType**(*cname, fname, pyname, octname*)

Holds strings for a certain datatype in different languages.

class diofant.utilities.codegen.**FCodeGen**(*project='project'*)

Generator for Fortran 95 code

The *.write()* method inherited from CodeGen will output a code file and an interface file, <prefix>.f90 and <prefix>.h respectively.

dump_f95(*routines, f, prefix, header=True, empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

- **routines** (*list*) – A list of Routine instances.
- **f** (*file-like*) – Where to write the file.
- **prefix** (*string*) – The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.
- **header** (*bool, optional*) – When True, a header comment is included on top of each source file. [default : True]
- **empty** (*bool, optional*) – When True, empty lines are included to structure the source files. [default : True]

dump_h(*routines, f, prefix, header=True, empty=True*)

Writes the interface to a header file.

This file contains all the function declarations.

Parameters

- **routines** (*list*) – A list of Routine instances.
- **f** (*file-like*) – Where to write the file.
- **prefix** (*string*) – The filename prefix.
- **header** (*bool, optional*) – When True, a header comment is included on top of each source file. [default : True]
- **empty** (*bool, optional*) – When True, empty lines are included to structure the source files. [default : True]

get_interface(*routine*)

Returns a string for the function interface.

The routine should have a single result object, which can be None. If the routine has multiple result objects, a CodeGenError is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

class diofant.utilities.codegen.**InputArgument**(*name, datatype=None, dimensions=None, precision=None*)

Input argument class.

class diofant.utilities.codegen.**OctaveCodeGen**(*project='project', cse=False*)

Generator for Octave code.

The .write() method inherited from CodeGen will output a code file <prefix>.m.

Octave .m files usually contain one function. That function name should match the file-name (prefix). If you pass multiple name_expr pairs, the latter ones are presumed to be private functions accessed by the primary function.

You should only pass inputs to argument_sequence: outputs are ordered according to their order in name_expr.

dump_m(*routines, f, prefix, header=True, empty=True, inline=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

- **routines** (*list*) – A list of Routine instances.
- **f** (*file-like*) – Where to write the file.
- **prefix** (*string*) – The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.
- **header** (*bool, optional*) – When True, a header comment is included on top of each source file. [default : True]
- **empty** (*bool, optional*) – When True, empty lines are included to structure the source files. [default : True]

routine(*name, expr, argument_sequence, global_vars=None*)

Specialized Routine creation for Octave.

class diofant.utilities.codegen.**OutputArgument**(*name, result_var, expr, datatype=None, dimensions=None, precision=None*)

OutputArgument are always initialized in the routine.

class diofant.utilities.codegen.**Result**(*expr, name=None, result_var=None, datatype=None, dimensions=None, precision=None*)

An expression for a return value.

The name result is used to avoid conflicts with the reserved word “return” in the python language. It is also shorter than ReturnValue.

These may or may not need a name in the destination (e.g., “return(x*y)” might return a value without ever naming it).

class diofant.utilities.codegen.**Routine**(*name, arguments, results, local_vars, global_vars*)

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or Fortran. Another example: Fortran and Python support complex numbers, while C does not.

property result_variables

Returns a list of OutputArgument, InOutArgument and Result.

If return values are present, they are at the end of the list.

property variables

Returns a set of all variables possibly used in the routine.

For routines with unnamed return values, the dummies that may or may not be used will be included in the set.

diofant.utilities.codegen.**codegen**(*name_expr, language, prefix=None, project='project', to_files=False, header=True, empty=True, argument_sequence=None, global_vars=None*)

Generate source code for expressions in a given language.

Parameters

- **name_expr** (*tuple, or list of tuples*) - A single (name, expression) tuple or a list of (name, expression) tuples. Each tuple corresponds to a routine. If the expression is an equality (an instance of class Equality) the left hand side is considered an output argument. If expression is an iterable, then the routine will have multiple outputs.
- **language** (*string*) - A string that indicates the source code language. This is case insensitive. Currently, 'C', 'F95' and 'Octave' are supported. 'Octave' generates code compatible with both Octave and Matlab.
- **prefix** (*string, optional*) - A prefix for the names of the files that contain the source code. Language-dependent suffixes will be appended. If omitted, the name of the first name_expr tuple is used.
- **project** (*string, optional*) - A project name, used for making unique pre-processor instructions. [default: "project"]
- **to_files** (*bool, optional*) - When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]
- **header** (*bool, optional*) - When True, a header is written on top of each source file. [default: True]
- **empty** (*bool, optional*) - When True, empty lines are used to structure the code. [default: True]
- **argument_sequence** (*iterable, optional*) - Sequence of arguments for the routine in a preferred order. A CodeGenError is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.
- **global_vars** (*iterable, optional*) - Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

Examples

```
>>> [(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) {  
    double f_result;  
    f_result = x + y*z;  
    return f_result;  
}  
>>> print(h_name)  
test.h  
>>> print(c_header)  
#ifndef PROJECT_TEST_H  
#define PROJECT_TEST_H  
double f(double x, double y, double z);  
#endif
```

Another example using Equality objects to give named outputs. Here the filename (prefix) is taken from the first (name, expr) pair.

```
>>> from diofant.abc import f, g
>>> [(c_name, c_code),
...   (h_name, c_header)] = codegen([('myfcn', x + y),
...                                   ('fcn2', [Eq(f, 2*x), Eq(g, y)])],
...                                   'C', header=False, empty=False)
>>> print(c_name)
myfcn.c
>>> print(c_code)
#include "myfcn.h"
#include <math.h>
double myfcn(double x, double y) {
    double myfcn_result;
    myfcn_result = x + y;
    return myfcn_result;
}
void fcn2(double x, double y, double *f, double *g) {
    (*f) = 2*x;
    (*g) = y;
}
```

If the generated function(s) will be part of a larger project where various global variables have been defined, the 'global_vars' option can be used to remove the specified variables from the function signature

```
>>> [(f_name, f_code), header] = codegen(
...     ('f', x+y*z), 'F95', header=False, empty=False,
...     argument_sequence=(x, y), global_vars=(z,))
>>> print(f_code)
REAL*8 function f(x, y)
implicit none
REAL*8, intent(in) :: x
REAL*8, intent(in) :: y
f = x + y*z
end function
```

`diofant.utilities.codegen.get_default_datatype(expr)`

Derives an appropriate datatype based on the expression.

`diofant.utilities.codegen.make_routine(name, expr, argument_sequence=None, global_vars=None, language='F95')`

A factory that makes an appropriate Routine from an expression.

Parameters

- **name** (*string*) – The name of this routine in the generated code.
- **expr** (*expression or list/tuple of expressions*) – A Diofant expression that the Routine instance will represent. If given a list or tuple of expressions, the routine will be considered to have multiple return values and/or output arguments.
- **argument_sequence** (*list or tuple, optional*) – List arguments for the routine in a preferred order. If omitted, the results are language dependent, for example, alphabetical order or in the same order as the given expressions.
- **global_vars** (*iterable, optional*) – Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.
- **language** (*string, optional*) – Specify a target language. The Routine itself should be language-agnostic but the precise way one is created, error checking, etc depend on the language. [default: "F95"].
- **A decision about whether to use output arguments or return values is made**

- **depending on both the language and the particular mathematical expressions.**
- **For an expression of type Equality, the left hand side is typically made**
- **into an OutputArgument (or perhaps an InOutArgument if appropriate).**
- **Otherwise, typically, the calculated expression is made a return values of**
- **the routine.**

Examples

```
>>> from diofant.abc import f, g
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars
set()
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x, 2]])])
>>> [arg.result_var for arg in r.results]
[result_...]
>>> [arg.expr for arg in r.results]
[x*y]
>>> [arg.name for arg in r.arguments]
[x, y, f, g, out_...]
```

We can examine the various arguments more closely:

```
>>> [a.name for a in r.arguments if isinstance(a, InputArgument)]
[x, y]
```

```
>>> [a.name for a in r.arguments if isinstance(a, OutputArgument)]
[f, out_...]
>>> [a.expr for a in r.arguments if isinstance(a, OutputArgument)]
[1, Matrix([[x, 2]])]
```

```
>>> [a.name for a in r.arguments if isinstance(a, InOutArgument)]
[g]
>>> [a.expr for a in r.arguments if isinstance(a, InOutArgument)]
[g + x]
```


4.18.3 Decorator

Useful utility decorators.

`diofant.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.

`diofant.utilities.decorator.doctest_depends_on(exe=None, modules=None, disable_viewers=None)`

Adds metadata about the dependencies which need to be met for doctesting the docstrings of the decorated objects.

`class diofant.utilities.decorator.no_attrs_in_subclass(cls, f)`

Don't 'inherit' certain attributes from a base class

```
>>> class A:
...     x = 'test'
```

```
>>> A.x = no_attrs_in_subclass(A, A.x)
```

```
>>> class B(A):
...     pass
```

```
>>> hasattr(A, 'x')
True
>>> hasattr(B, 'x')
False
```

4.18.4 Enumerative

This module includes functions and classes for enumerating and counting multiset partitions.

`diofant.utilities.enumerative.multiset_partitions_taocp(multiplicities)`

Enumerates partitions of a multiset.

Parameters

multiplicities – list of integer multiplicities of the components of the multiset.

Yields

state – Internal data structure which encodes a particular partition. This output is then usually processed by a visitor function which combines the information from this data structure with the components themselves to produce an actual partition.

Unless they wish to create their own visitor function, users will have little need to look inside this data structure. But, for reference, it is a 3-element list with components:

f

is a frame array, which is used to divide `pstack` into parts.

lpart

points to the base of the topmost part.

pstack

is an array of `PartComponent` objects.

The state output offers a peek into the internal data structures of the enumeration function. The client should treat this as read-only; any modification of the data structure will cause unpredictable (and almost certainly incorrect) results. Also, the components of `state` are modified in place at each iteration. Hence, the visitor must be called at each loop iteration. Accumulating the `state` instances and processing them later will not work.

Examples

```
>>> # variables components and multiplicities represent the multiset 'abb'
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> [list_visitor(state, components) for state in states]
[[['a', 'b', 'b']],
 [['a', 'b'], ['b']],
 [['a'], ['b'], ['b']],
 [['a'], ['b'], ['b']]]
```

`diofant.utilities.enumerative.factoring_visitor(state, primes)`

Use with `multiset_partitions_taocp` to enumerate the ways a number can be expressed as a product of factors. For this usage, the exponents of the prime factors of a number are arguments to the partition enumerator, while the corresponding prime factors are input here.

Examples

To enumerate the factorings of a number we can think of the elements of the partition as being the prime factors and the multiplicities as being their exponents.

```
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> [factoring_visitor(state, primes) for state in states]
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]
```

`diofant.utilities.enumerative.list_visitor(state, components)`

Return a list of lists to represent the partition.

Examples

```
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc') # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3]) # for multiset '1 2 2 3'
[[1, 2, 2], [3]]
```

The approach of the function `multiset_partitions_taocp` is extended and generalized by the class `MultisetPartitionTraverser`.

class diofant.utilities.enumerative.MultisetPartitionTraverser

Has methods to enumerate and count the partitions of a multiset.

This implements a refactored and extended version of Knuth's algorithm 7.1.2.5M.

The enumeration methods of this class are generators and return data structures which can be interpreted by the same visitor functions used for the output of `multiset_partitions_taocp`.

See also:

[`multiset_partitions_taocp`](#) (page 725)

Examples

```
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4, 4, 4, 2])
127750
>>> m.count_partitions([3, 3, 3])
686
```

References

- Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.
- On a Problem of Oppenheim concerning “Factorisatio Numerorum” E. R. Canfield, Paul Erdős, Carl Pomerance, JOURNAL OF NUMBER THEORY, Vol. 17, No. 1. August 1983. See section 7 for a description of an algorithm similar to Knuth's.
- Generating Multiset Partitions, Brent Yorgey, The Monad.Reader, Issue 8, September 2007.

count_partitions(*multiplicities*)

Returns the number of partitions of a multiset whose components have the multiplicities given in *multiplicities*.

For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the `MultisetPartitionTraverser` object and persists across calls. If the the user does not expect to call `count_partitions` for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to `count_partitions`, so it may be advantageous not to clear the object.

Examples

```
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9, 8, 2])
288716
>>> m.count_partitions([2, 2])
9
>>> del m
```

Notes

If one looks at the workings of Knuth’s algorithm M, it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

- 1) This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)
- 2) Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no `_range` version of `count_partitions`. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

References

- Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.

enum_all(*multiplicities*)

Enumerate the partitions of a multiset.

See also:

which provides the same result as this method, but is about twice as fast. Hence, `enum_all` is primarily useful for testing. Also see the function `enum_states` for a discussion of states and visitors.

Enumerate the partitions of a multiset with $lb < \text{num}(\text{parts})$

enum_all (page 728), *enum_small* (page 730), *enum_range* (page 729)

- **multiplicities** – list of multiplicities of the components of the multiset.
- **lb** – Number of parts in the partition must be greater than this lower bound.

In particular, if partitions with exactly k parts are desired, call with (multiplicities, $k - 1, k$). This method generalizes `enum_all`, `enum_small`, and `enum_large`.

Examples

```
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2, 2], 1, 2)
>>> [list_visitor(state, 'ab') for state in states]
[[['a', 'a'], ['b'], ['b']],
 [['a', 'a'], ['b'], ['b']],
 [['a', 'b'], ['b'], ['a']],
 [['a', 'b'], ['a'], ['b']]]
```

enum_small(multiplicities, ub)

Enumerate multiset partitions with no more than ub parts.

Equivalent to `enum_range(multiplicities, 0, ub)`

See also:

[enum_all](#) (page 728), [enum_large](#) (page 729), [enum_range](#) (page 729)

Parameters

- **multiplicities** - list of multiplicities of the components of the multiset.
- **ub** - Maximum number of parts

Examples

```
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2, 2], 2)
>>> [list_visitor(state, 'ab') for state in states]
[[['a', 'a'], ['b'], ['b']],
 [['a', 'a'], ['b'], ['b']],
 [['a', 'b'], ['b'], ['a']],
 [['a', 'b'], ['a'], ['b']]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth.

References

- Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.

4.18.5 Iterables

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.

`partitions`:

```
>>> from diofant.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 diofant.utilities.iterables import multiset_partitions
>>> list(multiset_partitions(3, 2))
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
>>> list(multiset_partitions([1, 1, 1, 2], 2))
[[[1, 1, 1], [2]], [[1, 1, 2], [1]], [[1, 1], [1, 2]]]
```

Docstring

class diofant.utilities.iterables.NotIterable

Use this as mixin when creating a class which is not supposed to return true when `is_iterable()` is called on its instances. I.e. avoid infinite loop when calling e.g. `list()` on the instance

diofant.utilities.iterables.cantor_product(*args)

Breadth-first (diagonal) cartesian product of iterables.

Each iterable is advanced in turn in a round-robin fashion. As usual with breadth-first, this comes at the cost of memory consumption.

```
>>> from itertools import count, islice
>>> list(islice(cantor_product(count(), count()), 9))
[(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]
```

diofant.utilities.iterables.common_prefix(*seqs)

Return the subsequence that is a common start of sequences in `seqs`.

```
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([1, 2, 3], [1, 2, 5])
[1, 2]
>>> common_prefix([1, 2, 3], [1, 3, 5])
[1]
```

diofant.utilities.iterables.common_suffix(*seqs)

Return the subsequence that is a common ending of sequences in `seqs`.

```
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([1, 2, 3], [9, 2, 3])
[2, 3]
>>> common_suffix([1, 2, 3], [9, 7, 3])
[3]
```

diofant.utilities.iterables.default_sort_key(item, order=None)

Return a key that can be used for sorting.

The key has the structure:

(class_key, (len(args), args), exponent.sort_key(), coefficient)

This key is supplied by the `sort_key` routine of Basic objects when `item` is a Basic object or an object (other than a string) that sympifies to a Basic object. Otherwise, this function produces the key.

The `order` argument is passed along to the `sort_key` routine and is used to determine how the terms *within* an expression are ordered. (See examples below) order options

are: 'lex', 'grlex', 'grevlex', and reversed values of the same (e.g. 'rev-lex'). The default order value is None (which translates to 'lex').

Examples

```
>>> from diofant.core.function import UndefinedFunction
```

The following are equivalent ways of getting the key for an object:

```
>>> x.sort_key() == default_sort_key(x)
True
```

Here are some examples of the key that is produced:

```
>>> default_sort_key(UndefinedFunction('f'))
((0, 0, 'UndefinedFunction'), (1, ('f',)), ((1, 0, 'Number'),
(0, ()), (1, 1))
>>> default_sort_key('1')
((0, 0, 'str'), (1, ('1',)), ((1, 0, 'Number'), (0, ()), (1, 1))
>>> default_sort_key(Integer(1))
((1, 0, 'Number'), (0, ()), (1, 1))
>>> default_sort_key(2)
((1, 0, 'Number'), (0, ()), (2, 2))
```

While `sort_key` is a method only defined for Diofant objects, `default_sort_key` will accept anything as an argument so it is more robust as a sorting key. For the following, using `key= lambda i: i.sort_key()` would fail because 2 doesn't have a `sort_key` method; that's why `default_sort_key` is used. Note, that it also handles sympification of non-string items like ints:

```
>>> a = [2, I, -I]
>>> sorted(a, key=default_sort_key)
[2, -I, I]
```

The returned key can be used anywhere that a key can be specified for a function, e.g. `sort`, `min`, `max`, etc...:

```
>>> a.sort(key=default_sort_key)
>>> a[0]
2
>>> min(a, key=default_sort_key)
2
```

Notes

The key returned is useful for getting items into a canonical order that will be the same across platforms. It is not directly useful for sorting lists of expressions:

```
>>> a, b = x, 1/x
```

Since `a` has only 1 term, its value of `sort_key` is unaffected by order:

```
>>> a.sort_key() == a.sort_key('rev-lex')
True
```

If `a` and `b` are combined then the key will differ because there are terms that can be ordered:


```
>>> eq = a + b
>>> eq.sort_key() == eq.sort_key('rev-lex')
False
>>> eq.as_ordered_terms()
[x, 1/x]
>>> eq.as_ordered_terms('rev-lex')
[1/x, x]
```

But since the keys for each of these terms are independent of order's value, they don't sort differently when they appear separately in a list:

```
>>> sorted(eq.args, key=default_sort_key)
[1/x, x]
>>> sorted(eq.args, key=lambda i: default_sort_key(i, order='rev-lex'))
[1/x, x]
```

The order of terms obtained when using these keys is the order that would be obtained if those terms were *factors* in a product.

Although it is useful for quickly putting expressions in canonical order, it does not sort expressions based on their complexity defined by the number of operations, power of variables and others:

```
>>> sorted([sin(x)*cos(x), sin(x)], key=default_sort_key)
[sin(x)*cos(x), sin(x)]
>>> sorted([x, x**2, sqrt(x), x**3], key=default_sort_key)
[sqrt(x), x, x**2, x**3]
```

See also:

[ordered](#) (page 737), [diofant.core.expr.Expr.as_ordered_factors](#) (page 63), [diofant.core.expr.Expr.as_ordered_terms](#) (page 63)

`diofant.utilities.iterables.flatten(iterable, levels=None, cls=None)`

Recursively denest iterable containers.

```
>>> 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:

```
>>> class MyOp(Basic):
...     pass
>>> flatten([MyOp(1, MyOp(2, 3))], cls=MyOp)
[1, 2, 3]
```

adapted from https://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks

`diofant.utilities.iterables.group(seq, multiple=True)`

Splits a sequence into a list of lists of equal, adjacent elements.

Examples

```
>>> 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)]
```

See also:

[multiset](#) (page 735)

`diofant.utilities.iterables.is_iterable(i, exclude=(<class 'str'>, <class 'dict'>, <class 'diofant.utilities.iterables.NotIterable'>))`

Return a boolean indicating whether `i` is Diofant iterable. True also indicates that the iterator is finite, i.e. you e.g. call `list(...)` on the instance.

When Diofant 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](#) (page 734)

Examples

```
>>> things = [[1], (1,), {1}, Tuple(1), (j for j in [1, 2]), {1: 2}, '1', 1]
>>> for i in things:
...     print(f'{is_iterable(i)} {type(i)}')
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'diofant.core.containers.Tuple'>
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>
```

```
>>> is_iterable({}, exclude=None)
True
>>> is_iterable({}, exclude=str)
True
>>> is_iterable('no', exclude=str)
False
```

`diofant.utilities.iterables.is_sequence(i, include=None)`

Return a boolean indicating whether `i` is a sequence in the Diofant 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:

[is_iterable](#) (page 734)

Examples

```
>>> 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
```

`diofant.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

```
>>> 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'
```

`diofant.utilities.iterables.multiset(seq)`

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

Examples

```
>>> multiset('mississippi')
{'i': 4, 'm': 1, 'p': 2, 's': 4}
```

See also:

[group](#) (page 733)

`diofant.utilities.iterables.multiset_combinations(m, n, g=None)`

Return the unique combinations of size `n` from multiset `m`.

Examples

```
>>> from itertools import combinations
>>> [''.join(i) for i in multiset_combinations('baby', 3)]
['abb', 'aby', 'bby']
```

```
>>> def count(f, s):
...     return len(list(f(s, 3)))
```

The number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

```
>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)
```

`diofant.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.

Counting

The number of partitions of a set is given by the bell number:

```
>>> 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 diofant import Dummy, Sum, binomial, factorial
...     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.

Examples

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
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]]]
```

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.

See also:

[partitions](#) (page 740), [diofant.combinatorics.partitions.Partition](#) (page 142), [diofant.combinatorics.partitions.IntegerPartition](#) (page 144)

`diofant.utilities.iterables.multiset_permutations(m, size=None, g=None)`

Return the unique permutations of multiset *m*.

Examples

```
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

`diofant.utilities.iterables.numbered_symbols(prefix='x', cls=None, start=0, exclude=[], **assumptions)`

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in *exclude*.

Parameters

- **prefix** (*str*, *optional*) – The prefix to use. By default, this function will generate symbols of the form “x0”, “x1”, etc.
- **cls** (*class*, *optional*) – The class to use. By default, it uses `Symbol`, but you can also use `Wild` or `Dummy`.
- **start** (*int*, *optional*) – The start number. By default, it is 0.

Returns

sym (*Symbol*) – The subscripted symbols.

`diofant.utilities.iterables.ordered(seq, keys=None, default=True, warn=False)`

Return an iterator of the *seq* where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.

Two default keys will be applied if 1) keys are not provided or 2) the given keys don't resolve all ties (but only if `default` is `True`). The two keys are `nodes` (which places smaller expressions before large) and `defaultsortkey` which (if the `sortkey` for an object is defined properly) should resolve any ties.

If `warn` is `True` then an error will be raised if there were no keys remaining to break ties. This can be used if it was expected that there should be no ties between items that are not identical.

Examples

The `count_ops` is not sufficient to break ties in this list and the first two items appear in their original order (i.e. the sorting is stable):

```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3],
...              count_ops, default=False, warn=False))
[y + 2, x + 2, x**2 + y + 3]
```

The `default_sort_key` allows the tie to be broken:

```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3]))
[x + 2, y + 2, x**2 + y + 3]
```

Here, sequences are sorted by length, then sum:

```
>>> seq, keys = [[[1, 2, 1], [0, 3, 1], [1, 1, 3], [2], [1]],
...              [lambda x: len(x), lambda x: sum(x)]]
>>> list(ordered(seq, keys, default=False, warn=False))
[[1], [2], [1, 2, 1], [0, 3, 1], [1, 1, 3]]
```

If `warn` is `True`, an error will be raised if there were not enough keys to break ties:

```
>>> list(ordered(seq, keys, default=False, warn=True))
Traceback (most recent call last):
ValueError: not enough keys to break ties
```

Notes

The decorated sort is one of the fastest ways to sort a sequence for which special item comparison is desired: the sequence is decorated, sorted on the basis of the decoration (e.g. making all letters lower case) and then undecorated. If one wants to break ties for items that have the same decorated value, a second key can be used. But if the second key is expensive to compute then it is inefficient to decorate all items with both keys: only those items having identical first key values need to be decorated. This function applies keys successively only when needed to break ties. By yielding an iterator, use of the tie-breaker is delayed as long as possible.

This function is best used in cases when use of the first key is expected to be a good hashing function; if there are no unique hashes from application of a key then that key should not have been used. The exception, however, is that even if there are many collisions, if the first group is small and one does not need to process all items in the list then time will not be wasted sorting what one was not interested in. For example, if one were looking for the minimum in a list and there were several criteria used to define the sort order, then this function would be good at returning that quickly if the first group of candidates is small relative to the number of items being processed.

`diofant.utilities.iterables.ordered_partitions(n, m=None, sort=True)`

Generates ordered partitions of integer `n`.

Parameters

- `m` (*int or None, optional*) – By default (`None`) gives partitions of all sizes, else only those with size `m`. In addition, if `m` is not `None` then partitions are generated *in place* (see examples).
- `sort` (*bool, optional*) – Controls whether partitions are returned in sorted order (default) when `m` is not `None`; when `False`, the partitions are returned as fast as possible with elements sorted, but when `m|n` the partitions will not be in ascending lexicographical order.

Examples

All partitions of 5 in ascending lexicographical:

```
>>> for p in ordered_partitions(5):
...     print(p)
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 3]
[1, 2, 2]
[1, 4]
[2, 3]
[5]
```

Only partitions of 5 with two parts:

```
>>> for p in ordered_partitions(5, 2):
...     print(p)
[1, 4]
[2, 3]
```

When `m` is given, a given list objects will be used more than once for speed reasons so you will not see the correct partitions unless you make a copy of each as it is generated:

```
>>> list(ordered_partitions(7, 3))
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [2, 2, 2]]
>>> [list(p) for p in ordered_partitions(7, 3)]
[[1, 1, 5], [1, 2, 4], [1, 3, 3], [2, 2, 3]]
```

When `n` is a multiple of `m`, the elements are still sorted but the partitions themselves will be *unordered* if `sort` is `False`; the default is to return them in ascending lexicographical order.

```
>>> for p in ordered_partitions(6, 2):
...     print(p)
[1, 5]
[2, 4]
[3, 3]
```

But if speed is more important than ordering, `sort` can be set to `False`:

```
>>> for p in ordered_partitions(6, 2, sort=False):
...     print(p)
[1, 5]
[3, 3]
[2, 4]
```

References

- Generating Integer Partitions, [online], Available: <http://jeromekelleher.net/generating-integer-partitions.html>
- Jerome Kelleher and Barry O’Sullivan, “Generating All Partitions: A Comparison Of Two Encodings”, [online], Available: <https://arxiv.org/pdf/0909.2331v2.pdf>

`diofant.utilities.iterables.partitions(n, m=None, k=None, size=False)`

Generate all partitions of positive integer, n.

Parameters

- `m` (*integer (default gives partitions of all sizes)*) – limits number of parts in partition (mnemonic: m, maximum parts). Default value, None, gives partitions from 1 through n.
- `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”).**

Examples

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```
>>> from diofant.utilities.iterables import partitions
```

```
>>> for p in partitions(6, k=2):
...     print(p)
{2: 3}
{2: 2, 1: 2}
{2: 1, 1: 4}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m:

```
>>> for p in partitions(6, m=2):
...     print(p)
{6: 1}
{5: 1, 1: 1}
{4: 1, 2: 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.

```
>>> list(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})]
```

References

- <http://code.activestate.com/recipes/218332-generator-for-integer-partitions/>

Notes

Modified from Tim Peter's version to allow for k and m values.

See also:

diofant.combinatorics.partitions.Partition (page 142), *diofant.combinatorics.partitions.IntegerPartition* (page 144)

`diofant.utilities.iterables.permute_signs(t)`

Return iterator in which the signs of non-zero elements of t are permuted.

Examples

```
>>> list(permute_signs((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2)]
```

`diofant.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** (*diofant expression*) – The expression to traverse.
- **keys** ((*default None*) *sort key(s)*) – The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to `ordered()` as the only key(s) to use to sort the arguments; if key is simply True then the default keys of `ordered` will be used (node count and `default_sort_key`).

Yields

subtree (*diofant expression*) – All of the subtrees in the tree.

Examples

```
>>> from diofant.abc import w
```

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, keys=True))  
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

`diofant.utilities.iterables.rotate_left(x, y)`

Left rotates a list x by the number of steps specified in y.

Examples

```
>>> a = [0, 1, 2]  
>>> rotate_left(a, 1)  
[1, 2, 0]
```

`diofant.utilities.iterables.rotate_right(x, y)`

Right rotates a list x by the number of steps specified in y.

Examples

```
>>> a = [0, 1, 2]  
>>> rotate_right(a, 1)  
[2, 0, 1]
```

`diofant.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

```
>>> import operator  
>>> 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=operator.ge)  
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

`diofant.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.

Examples

```
>>> from collections import defaultdict
```

```
>>> sift(range(5), lambda x: x % 2) == defaultdict(int, {0: [0, 2, 4], 1: [1, 3]})
True
```

sift() returns a defaultdict() object, so any key that has no matches will give [].

```
>>> dl = sift([x], lambda x: x.is_commutative)
>>> dl == defaultdict(list, {True: [x]})
True
>>> dl[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]) ==
...   defaultdict(list, {E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}))
True
```

If you need to sort the sifted items it might be better to use ordered which can economically apply multiple sort keys to a sequence while sorting.

See also:

[ordered](#) (page 737)

diofant.utilities.iterables.**signed_permutations**(t)

Return iterator in which the signs of non-zero elements of t and the order of the elements are permuted.

Examples

```
>>> list(signed_permutations((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2), (0, 2, 1),
 (0, -2, 1), (0, 2, -1), (0, -2, -1), (1, 0, 2), (-1, 0, 2),
 (1, 0, -2), (-1, 0, -2), (1, 2, 0), (-1, 2, 0), (1, -2, 0),
 (-1, -2, 0), (2, 0, 1), (-2, 0, 1), (2, 0, -1), (-2, 0, -1),
 (2, 1, 0), (-2, 1, 0), (2, -1, 0), (-2, -1, 0)]
```

diofant.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

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”:

```
>>> from diofant.utilities.iterables import subsets
```

```
>>> 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)]
```

`diofant.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`.

`diofant.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

```
>>> 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]]
```

4.18.6 Lambdify

This module provides convenient functions to transform diofant expressions to lambda functions which can be used to calculate numerical values very fast.

`diofant.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 Diofant 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 diofant function, attach implementation to it.
- **implementation** (*callable*) – numerical implementation to be called by `evalf()` or `lambdify`

Returns

afunc (*diofant.FunctionClass instance*) – function with attached implementation

Examples

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> lam_f = lambdify(x, f(x))
>>> lam_f(4)
5
```

`diofant.utilities.lambdify.lambdastr(args, expr, printer=None, dummify=False)`
Returns a string that can be evaluated to a lambda function.

Examples

```
>>> lambdastr(x, x**2)
'lambda x: (x**2)'
>>> lambdastr((x, y, z), [z, y, x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, `lambdastr` will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))(*list(__flatten_args__([_0,_1])))'
```

`diofant.utilities.lambdify.lambdify(args, expr, modules=None, printer=None, useimps=True, dummify=True)`

Returns a lambda function for fast calculation of numerical values.

If not specified differently by the user, `modules` defaults to `["numpy"]` if NumPy is installed, and `["math", "mpmath", "sympy"]` if it isn't, that is, Diofant functions are replaced as far as possible by either numpy functions if available, and Python's standard library `math`, or `mpmath` functions otherwise. To change this behavior, the “modules” argument can be used. It accepts:

- the strings “math”, “mpmath”, “numpy”, “diofant”
- any modules (e.g. `math`)
- dictionaries that map names of diofant functions to arbitrary functions
- lists that contain a mix of the arguments above, with higher priority given to entries appearing first.

The default behavior is to substitute all arguments in the provided expression with dummy symbols. This allows for applied functions (e.g. `f(t)`) to be supplied as arguments. Call the function with `dummify=False` if dummy substitution is unwanted (and

args is not a string). If you want to view the lambdified function or provide “diofant” as the module, you should probably set `dummify=False`.

In previous releases `lambdify` replaced `Matrix` with `numpy.matrix` by default. As of release 0.7.7 `numpy.array` is the default. To get the old default behavior you must pass in `[{'ImmutableMatrix': numpy.matrix}, 'numpy']` to the modules kwarg.

- (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', 'diofant'))
```

- (2) Use some other module:

```
>>> import numpy
>>> f = lambdify((x, y), tan(x*y), numpy)
```

Attention: There are naming differences between numpy and diofant. So if you simply take the numpy module, e.g. `diofant.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 a dictionary defining custom functions:

```
>>> def my_cool_function(x):
...     return f'sin({x}) is cool'
>>> myfuncs = {'sin': my_cool_function}
>>> f = lambdify(x, sin(x), myfuncs)
>>> f(1)
'sin(1) is cool'
```

Examples

```
>>> from diofant.abc import w
```

```
>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1, 2, 3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='diofant')
>>> row(1, 2)
Matrix([[1, 3]])
```

Tuple arguments are handled and the lambdified function should be called with the same type of arguments as were used to create the function.:

```
>>> f = lambdify((x, (y, z)), x + y)
>>> f(1, (2, 4))
3
```

A more robust way of handling this is to always work with flattened arguments:

```
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```

Functions present in *expr* can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. Usually you attach this using the `implemented_function` factory:

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

`lambdify` always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_imps` input parameter is `False`.

4.18.7 Memoization

`diofant.utilities.memoization.recurrence_memo(initial)`

Memo decorator for sequences defined by recurrence

See usage examples e.g. in the `specfun/combinatorial` module

4.18.8 Miscellaneous

Miscellaneous stuff that doesn't really fit anywhere else.

`diofant.utilities.misc.filldedent(s, w=70)`

Strips leading and trailing empty lines from a copy of *s*, then dedents, fills and returns it.

Empty line stripping serves to deal with docstrings like this one that start with a newline after the initial triple quote, inserting an empty line at the beginning of the string.

4.18.9 Randomised Testing

Helpers for randomized testing.

`diofant.utilities.randtest.random_complex_number(a=2, b=-1, c=3, d=1, rational=True)`

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee $b \leq \text{Im } z \leq d$, $a \leq \text{Re } z \leq c$

```
diofant.utilities.randtest.verify_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

```
>>> verify_derivative_numerically(sin(x), x)
true
```

```
diofant.utilities.randtest.verify_numerically(f, g, z=None, tol=1e-06, a=2, b=-1,
                                              c=3, d=1)
```

Test numerically that f and g agree when evaluated in the argument z .

If z is `None`, all symbols will be tested. This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> verify_numerically(sin(x)**2 + cos(x)**2, 1, x)
true
```

4.19 Parsing

4.19.1 Parsing Functions Reference

```
diofant.parsing.sympy_parser.parse_expr(s, local_dict=None,
                                         transformations=(<function
                                                         lambda_notation>, <function auto_symbol>,
                                                         <function auto_number>),
                                         global_dict=None, evaluate=True)
```

Converts the string s to a Diofant 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 diofant 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 Diofant equivalents, convert undefined variables into Diofant symbols.

- **evaluate** (*bool, optional*) - When False, the order of the arguments will remain as they were in the string and automatic simplification that would normally occur is suppressed. (see examples)

Examples

```
>>> parse_expr('1/2')
1/2
>>> type(_)
<class 'diofant.core.numbers.Half'>
>>> transformations = (standard_transformations +
...                    (implicit_multiplication_application,))
>>> parse_expr('2x', transformations=transformations)
2*x
```

When evaluate=False, some automatic simplifications will not occur:

```
>>> parse_expr('2**3'), parse_expr('2**3', evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows one to tell exactly how the expression was entered:

```
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

See also:

[diofant.parsing.sympy_parser.stringify_expr](#) (page 749), [diofant.parsing.sympy_parser.eval_expr](#) (page 749), [diofant.parsing.sympy_parser.standard_transformations](#) (page 750), [diofant.parsing.sympy_parser.implicit_multiplication_application](#) (page 751)

`diofant.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.

`diofant.parsing.sympy_parser.eval_expr(code, local_dict, global_dict)`

Evaluate Python code generated by `stringify_expr`.

Generally, `parse_expr` should be used.

`diofant.parsing.maxima.parse_maxima(str, globals=None, name_dict={})`

`diofant.parsing.mathematica.mathematica(s)`

4.19.2 Parsing Transformations Reference

A transformation is a function that accepts the arguments `tokens`, `local_dict`, `global_dict` and returns a list of transformed tokens. They can be used by passing a list of functions to `parse_expr()` (page 748) and are applied in the order given.

`diofant.parsing.sympy_parser.standard_transformations = (<function lambda_notation>, <function auto_symbol>, <function auto_number>)`

Standard transformations for `parse_expr()` (page 748). Inserts calls to `Symbol` (page 80), `Integer` (page 89), and other Diofant datatypes.

`diofant.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`.

`diofant.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:

```
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...         return tokenSplittable(symbol)
...     return False
>>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations +
...            (transformation, implicit_multiplication))
unsplittable
```

`diofant.parsing.sympy_parser.implicit_multiplication(result, local_dict, global_dict)`

Makes the multiplication operator optional in most cases.

Use this before `implicit_application()` (page 750), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> transformations = standard_transformations + (implicit_multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y
```

`diofant.parsing.sympy_parser.implicit_application(result, local_dict, global_dict)`

Makes parentheses optional in some cases for function calls.

Use this after `implicit_multiplication()` (page 750), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)
```

`diofant.parsing.sympy_parser.function_exponentiation(tokens, local_dict, global_dict)`

Allows functions to be exponentiated, e.g. $\cos^{**2}(x)$.

Examples

```
>>> transformations = standard_transformations + (function_exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

`diofant.parsing.sympy_parser.implicit_multiplication_application(result, local_dict, global_dict)`

Allows a slightly relaxed syntax.

- Parentheses for single-argument method calls are optional.
- Multiplication is implicit.
- Symbol names can be split (i.e. spaces are not needed between symbols).
- Functions can be exponentiated.

Examples

```
>>> 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)
```

`diofant.parsing.sympy_parser.rationalize(tokens, local_dict, global_dict)`

Converts floats into Rational. Run AFTER `auto_number`.

`diofant.parsing.sympy_parser.convert_xor(tokens, local_dict, global_dict)`

Treats XOR, `^`, as exponentiation, `**`.

These are included in `:data:diofant.parsing.sympy_parser.standard_transformations` and generally don't need to be manually added by the user.

`diofant.parsing.sympy_parser.auto_symbol(tokens, local_dict, global_dict)`

Inserts calls to Symbol/Function for undefined variables.

`diofant.parsing.sympy_parser.auto_number(tokens, local_dict, global_dict)`

Converts numeric literals to use Diofant equivalents.

Complex numbers use `I`; integer literals use `Integer`, float literals use `Float`, and repeating decimals use `Rational`.

4.20 Calculus

Some calculus-related methods waiting to find a better place in the Diofant modules tree.

`diofant.calculus.singularities.singularities(f, x)`

Find singularities of real-valued function f with respect to x .

Examples

```
>>> singularities(1/(1 + x), x)
{-1}
```

```
>>> singularities(exp(1/x) + log(x + 1), x)
{-1, 0}
```

```
>>> singularities(exp(1/log(x + 1)), x)
{0}
```

Notes

Removable singularities are not supported now.

References

- https://en.wikipedia.org/wiki/Mathematical_singularity

class `diofant.calculus.limits.Limit(e, z, z0, dir=None)`

Represents an unevaluated directional limit of `expr` at the point `z0`.

Parameters

- **expr** (*Expr*) – algebraic expression
- **z** (*Symbol*) – variable of the `expr`
- **z0** (*Expr*) – limit point, z_0
- **dir** (*Expr or Reals, optional*) – selects the direction (as `sign(dir)`) to approach the limit point if the `dir` is an `Expr`. For infinite `z0`, the default value is determined from the direction of the infinity (e.g., the limit from the left, `dir=1`, for ∞). Otherwise, the default is the limit from the right, `dir=-1`. If `dir=Reals`, the limit is the bidirectional real limit.

Examples

```
>>> Limit(1/x, x, 0, dir=1)
Limit(1/x, x, 0, dir=1)
>>> .doit()
-oo
```

See also:

[limit](#) (page 753)

doit(***hints*)

Evaluates limit.

Notes

First we handle some trivial cases (i.e. constant), then try Gruntz algorithm (see the [gruntz](#) (page 787) module).

`diofant.calculus.limits.limit(expr, z, z0, dir=None, **kwargs)`

Compute the directional limit of *expr* at the point *z0*.

Examples

```
>>> limit(1/x, x, 0)
oo
>>> limit(1/x, x, 0, dir=1)
-oo
>>> limit(1/x, x, oo)
0
```

See also:

[Limit](#) (page 752)

`diofant.calculus.optimization.maximize(f, *v)`

Maximizes *f* with respect to given variables *v*.

See also:

[minimize](#) (page 753)

`diofant.calculus.optimization.minimize(f, *v)`

Minimizes *f* with respect to given variables *v*.

Examples

```
>>> minimize(x**2, x)
(0, {x: 0})
```

```
>>> minimize([x**2, x >= 1], x)
(1, {x: 1})
>>> minimize([-x**2, x >= -2, x <= 1], x)
(-4, {x: -2})
```

See also:

[maximize](#) (page 753)

`diofant.calculus.order.0`

alias of [Order](#) (page 754)

class `diofant.calculus.order.Order(expr, var=None, point=0, **kwargs)`

Represents the limiting behavior of function.

The formal definition for order symbol $O(f(x))$ (Big O) is that $g(x) \in O(f(x))$ as $x \rightarrow a$ iff

$$\limsup_{x \rightarrow a} \left| \frac{g(x)}{f(x)} \right| < \infty$$

Parameters

- **expr** (*Expr*) – an expression
- **var** (*Symbol, optional*) – a variable of the expr. If not provided, the expression is assumed to be univariate and it's variable is used.
- **point** (*Expr, optional*) – a limit point, default is zero.

Examples

The order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example, $O(x^3)$ corresponds to any terms proportional to x^3, x^4, \dots and any higher power:

```
>>> 1 + x + x**2 + x**3 + x**4 + 0(x**3)
1 + x + x**2 + 0(x**3)
```

$O(f(x))$ is automatically transformed to `0(f(x)).as_leading_term(x)`:

```
>>> 0(x + x**2)
0(x)
>>> 0(cos(x))
0(1, x)
```

Some arithmetic operations:

```
>>> 0(x)*x
0(x**2)
>>> 0(x) - 0(x)
0(x)
```

The Big O symbol is a set, so we support membership test:

```
>>> x in 0(x)
True
>>> 0(x) in 0(1, x)
True
>>> 0(x**2) in 0(x)
True
```

Limit points other than zero are also supported:

```
>>> 0(x) == 0(x, x, 0)
True
>>> 0(x + x**2, x, oo)
0(x**2, x, oo)
>>> 0(cos(x), x, pi/2)
0(x - pi/2, x, pi/2)
```

References

- https://en.wikipedia.org/wiki/Big_O_notation

contains(*expr*)

Membership test.

Returns

Boolean or None - Return True if *expr* belongs to self. Return False if self belongs to *expr*. Return None if the inclusion relation cannot be determined.

`diofant.calculus.residues.residue(expr, x, x0)`

Finds the residue of *expr* at the point $x=x_0$.

The residue is defined as the coefficient of $1/(x-x_0)$ in the power series expansion around $x = x_0$.

This notion is essential for the Residue Theorem.

Examples

```
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

References

- https://en.wikipedia.org/wiki/Residue_%28complex_analysis%29
- https://en.wikipedia.org/wiki/Residue_theorem

INTERNALS

This section covers the developers-only documentation, i.e part of the API that may be changed anytime by anyone.

5.1 Internals of the Polynomial Manipulation Module

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* (page 426). 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))`.

5.1.1 Manipulation of sparse, distributed polynomials

Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a *sparse* representation.

Sparse polynomials are represented as dictionaries.

```
diofant.polys.rings.ring(symbols, domain, order=<diofant.polys.orderings.LexOrder  
                        object>)
```

Construct a polynomial ring returning (ring, x_1 , ..., x_n).

Parameters

- **symbols** (*str*, *Symbol/Expr* or sequence of *str*, *Symbol/Expr* (non-empty))
- **domain** (*Domain* (page 427) or coercible)
- **order** (*Order* (page 785) or coercible, optional, defaults to `lex`)

Examples

```
>>> R, x, y, z = ring('x y z', ZZ)
>>> R
ZZ[x, y, z]
>>> x + y + z
x + y + z
```

class diofant.polys.rings.PolyElement

Represent a polynomial in a multivariate polynomial ring.

A polynomial is mutable, until its hash is computed, e.g. for using an instance as a dictionary key.

See also:

[PolynomialRing](#) (page 429)

__add__(*other*)

Add two polynomials.

__eq__(*other*)

Equality test for polynomials.

Examples

```
>>> x, y = ring('x y', ZZ)
>>> p1 = (x + y)**2 + (x - y)**2
>>> p1 == 4*x*y
False
>>> p1 == 2*(x**2 + y**2)
True
```

__getitem__(*monom*, /)

Return the coefficient for the given monomial.

Parameters

monom (Monomial or PolyElement (with `is_monomial = True`) or 1)

Examples

```
>>> x, y, z = ring('x y z', ZZ)
>>> f = 3*x**2*y - x*y*z + 7*z**3 + 23
```

```
>>> f[x**2*y]
3
>>> f[x*y]
0
>>> f[1]
23
```

__mul__(*other*)

Multiply two polynomials.

__pow__(*n*, *mod=None*)

Raise polynomial to power *n*.

__radd__(*other*)

Add two polynomials.

`__rmul__(other)`

Multiply two polynomials.

`__rsub__(other)`

Subtract self from other, with other convertible to the coefficient domain.

`__setitem__(monom, coeff, /)`

Set the coefficient for the given monomial.

Parameters

- **monom** (Monomial or PolyElement (with `is_monomial = True`) or 1)
- **coeff** (*DomainElement*)

Examples

```
>>> _, x, y = ring('x y', ZZ)
```

```
>>> p = (x + y)**2
>>> p1 = p.copy()
>>> p2 = p
```

```
>>> p[x*y] = 0
>>> p1
x**2 + 2*x*y + y**2
>>> p2
x**2 + y**2
```

```
>>> _ = hash(p)
>>> p[x*y] = 1
Traceback (most recent call last):
RuntimeError: Polynomial x**2 + y**2 can't be modified
```

`__sub__(other)`

Subtract polynomial other from self.

`__weakref__`

list of weak references to the object (if defined)

`cancel(g, include=True)`

Cancel common factors in a rational function f/g .

Examples

```
>>> _, x, y = ring('x y', ZZ)
```

```
>>> (2*x**2 - 2).cancel(x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

`compose(x, a=None)`

Computes the functional composition.

`content()`

Returns GCD of polynomial's coefficients.

copy()

Return a shallow copy of self.

degree(x=0)

The leading degree in x or the main variable.

Note that the degree of 0 is negative floating-point infinity.

diff(x=0, m=1)

Computes partial derivative in x.

Examples

```
>>> R, x, y = ring('x y', ZZ)
>>> p = x + x**2*y**3
>>> p.diff(x)
2*x*y**3 + 1
```

discriminant()

Computes discriminant of a polynomial.

div(fv)

Division algorithm for multivariate polynomials.

Parameters

fv (*sequence of PolyElement's*) - List of divisors.

Returns

(qv, r) (*tuple*) - Where qv is the sequence of quotients and r is the remainder.

Notes

For multivariate polynomials the remainder is not uniquely determined, unless divisors form a Gröbner basis.

Examples

```
>>> R, x, y = ring('x y', ZZ)
>>> f = x**2*y
>>> f1, f2 = x**2 - y, x*y - 1
>>> f.div([f1, f2])
([y, 0], y**2)
>>> f.div([f2, f1])
([x, 0], x)
```

```
>>> g1, g2 = x - y**2, y**3 - 1
>>> f.div([g1, g2])[1] == f.div([g2, g1])[1]
True
```

References

- [CLOShea15], p. 64.

`gcdex(other)`

Extended Euclidean algorithm in $F[x]$.

Returns (s, t, h) such that $h = \gcd(\text{self}, \text{other})$ and $s*\text{self} + t*\text{other} = h$.

Examples

```
>>> _, x = ring('x', QQ)
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> f.gcdex(g)
(-1/5*x + 3/5, 1/5*x**2 - 6/5*x + 2, x + 1)
```

`half_gcdex(other)`

Half extended Euclidean algorithm in $F[x]$.

Returns (s, h) such that $h = \gcd(\text{self}, \text{other})$ and $s*\text{self} = h \pmod{\text{other}}$.

Examples

```
>>> _, x = ring('x', QQ)
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> f.half_gcdex(g)
(-1/5*x + 3/5, x + 1)
```

`integrate(x=0, m=1)`

Computes indefinite integral in x .

`leading_expv(order=None)`

Leading monomial tuple according to the monomial ordering.

Examples

```
>>> _, x, y, z = ring('x y z', ZZ)
>>> p = x**4 + x**3*y + x**2*z**2 + z**7
>>> p.leading_expv()
(4, 0, 0)
```

`leading_term(order=None)`

Leading term as a polynomial element.

Examples

```
>>> _, x, y = ring('x y', ZZ)
>>> (3*x*y + y**2).leading_term()
3*x*y
```

monic()

Divides all coefficients by the leading coefficient.

prem(*other*)

Polynomial pseudo-remainder.

Examples

```
>>> _, x, y = ring('x y', ZZ)
```

```
>>> (x**2 + x*y).prem(2*x + 2)
-4*y + 4
```

References

- [Knu85], p. 407.

primitive()

Returns content and a primitive polynomial.

resultant(*other*, *includePRS=False*)

Computes resultant of two polynomials in $K[X]$.

Examples

```
>>> _, x, y = ring('x y', ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> f.resultant(g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

subresultants(*other*)

Computes subresultant PRS of two polynomials in $K[X]$.

Examples

```
>>> _, 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
```

```
>>> f.subresultants(g) == [f, g, a, b]
True
```

tail_degree(x=0)

The tail degree in x or the main variable.

Note that the degree of 0 is negative floating-point infinity.

total_degree()

Returns the total degree.

class diofant.polys.univar.UnivarPolyElement

Element of univariate distributed polynomial ring.

decompose()

Compute 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. $f \circ g = f(x + b) \circ (g - b)$

2. $x^{**n} \circ x^{**m} = x^{**m} \circ x^{**n}$

3. $T_n \circ T_m = T_m \circ T_n$

where T_n and T_m are Chebyshev polynomials.

Examples

```
>>> _, x = ring('x', ZZ)
```

```
>>> (x**4 - 2*x**3 + x**2).decompose()
[x**2, x**2 - x]
```

References

- [KL89]

`sturm()`

Computes the Sturm sequence of f in $F[x]$.

Given a univariate, square-free polynomial $f(x)$ returns the associated Sturm sequence (see e.g. [DST88]) $f_0(x), \dots, f_n(x)$ defined by:

$$\begin{aligned} f_0(x), f_1(x) &= f(x), f'(x) \\ f_n &= -\text{rem}(f_{n-2}(x), f_{n-1}(x)) \end{aligned}$$

Examples

```
>>> _, x = ring('x', QQ)
```

```
>>> (x**3 - 2*x**2 + x - 3).sturm()
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2/9*x + 25/9, -2079/4]
```

5.1.2 Polynomial factorization algorithms

Many variants of Euclid's algorithm:

Classical remainder sequence

Let K be a field, and consider the ring $K[X]$ of polynomials in a single indeterminate X with coefficients in K . Given two elements f and g of $K[X]$ with $g \neq 0$ there are unique polynomials q and r such that $f = qg + r$ and $\deg(r) < \deg(g)$ or $r = 0$. They are denoted by $\text{quo}(f, g)$ (*quotient*) and $\text{rem}(f, g)$ (*remainder*), so we have the *division identity*

$$f = \text{quo}(f, g)g + \text{rem}(f, g).$$

It follows that every ideal I of $K[X]$ is a principal ideal, generated by any element $\neq 0$ of minimum degree (assuming I non-zero). In fact, if g is such a polynomial and f is any element of I , $\text{rem}(f, g)$ belongs to I as a linear combination of f and g , hence must be zero; therefore f is a multiple of g .

Using this result it is possible to find a **greatest common divisor** (gcd) of any polynomials f, g, \dots in $K[X]$. If I is the ideal formed by all linear combinations of the given polynomials with coefficients in $K[X]$, and d is its generator, then every common divisor of the polynomials also divides d . On the other hand, the given polynomials are multiples of the generator d ; hence d is a gcd of the polynomials, denoted $\text{gcd}(f, g, \dots)$.

An algorithm for the gcd of two polynomials f and g in $K[X]$ can now be obtained as follows. By the division identity, $r = \text{rem}(f, g)$ is in the ideal generated by f and g , as well as f is in the ideal generated by g and r . Hence the ideals generated by the pairs (f, g) and (g, r) are the same. Set $f_0 = f$, $f_1 = g$, and define recursively $f_i = \text{rem}(f_{i-2}, f_{i-1})$ for $i \geq 2$. The recursion ends after a finite number of steps with $f_{k+1} = 0$, since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs (f_{i-1}, f_i) generate the same ideal. In particular, the ideal generated by f and g is generated by f_k alone as $f_{k+1} = 0$. Hence $d = f_k$ is a gcd of f and g .

The sequence of polynomials f_0, f_1, \dots, f_k is called the *Euclidean polynomial remainder sequence* determined by (f, g) because of the analogy with the classical [Euclidean algorithm](#) for the gcd of natural numbers.

The algorithm may be extended to obtain an expression for d in terms of f and g by using the full division identities to write recursively each f_i as a linear combination of f and g . This leads to an equation

$$d = uf + vg \quad (u, v \in K[X])$$

analogous to [Bézout's identity](#) in the case of integers.

Simplified remainder sequences

Assume, as is usual, that the coefficient field K is the field of fractions of an integral domain A . In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If A is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in $K[X]$ is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial f in $K[X]$ can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation $f = cF$ where $c \in K$ is the *content* of f and F is a *primitive* polynomial, i.e., a polynomial in $A[X]$ with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials f and g with their primitive parts. This will only modify $\text{rem}(f, g)$ by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive $\text{gcd}(f, g)$.

This sequence is the *primitive polynomial remainder sequence*. It is an example of *general polynomial remainder sequences* where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

Subresultant sequence

The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [[Col67](#)] realized that the so-called *subresultant polynomials* of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in the subdomain A , so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in A .

To see how subresultants are associated with remainder sequences recall that all polynomials h in the sequence are linear combinations of the given polynomials f and g

$$h = uf + vg$$

with polynomials u and v in $K[X]$. Moreover, as is seen from the extended Euclidean algorithm, the degrees of u and v are relatively low, with limited growth from step to step.

Let $n = \deg(f)$, and $m = \deg(g)$, and assume $n \geq m$. If $\deg(h) = j < m$, the coefficients of the powers X^k ($k > j$) in the products uf and vg cancel each other. In particular, the products must have the same degree, say, l . Then $\deg(u) = l - n$ and $\deg(v) = l - m$ with a total of $2l - n - m + 2$ coefficients to be determined.

On the other hand, the equality $h = uf + vg$ implies that $l - j$ linear combinations of the coefficients are zero, those associated with the powers X^i ($j < i \leq l$), and one has a given non-zero value, namely the leading coefficient of h .

To satisfy these $l - j + 1$ linear equations the total number of coefficients to be determined cannot be lower than $l - j + 1$, in general. This leads to the inequality $l \geq n + m - j - 1$. Taking $l = n + m - j - 1$, we obtain $\deg(u) = m - j - 1$ and $\deg(v) = n - j - 1$.

In the case $j = 0$ the matrix of the resulting system of linear equations is the [Sylvester matrix](#) $S(f, g)$ associated to f and g , an $(n + m) \times (n + m)$ matrix with coefficients of f and g as entries. Its determinant is the [resultant](#) $\text{res}(f, g)$ of the pair (f, g) . It is non-zero if and only if f and g are relatively prime.

For any j in the interval from 0 to m the matrix of the linear system is an $(n + m - 2j) \times (n + m - 2j)$ submatrix of the Sylvester matrix. Its determinant $s_j(f, g)$ is called the j th *scalar subresultant* of f and g .

If $s_j(f, g)$ is not zero, the associated equation $h = uf + vg$ has a unique solution where $\deg(h) = j$ and the leading coefficient of h has any given value; the one with leading coefficient $s_j(f, g)$ is the j th *subresultant polynomial* or, briefly, *subresultant* of the pair (f, g) , and denoted $S_j(f, g)$. This choice guarantees that the remaining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if f and g are in $A[X]$, so is $S_j(f, g)$ as well. This construction of subresultants applies to any j between 0 and m regardless of the value of $s_j(f, g)$; if it is zero, then $\deg(S_j(f, g)) < j$.

The properties of subresultants are as follows. Let $n_0 = \deg(f)$, $n_1 = \deg(g)$, n_2, \dots, n_k be the decreasing sequence of degrees of polynomials in a remainder sequence. Let $0 \leq j \leq n_1$; then

- $s_j(f, g) \neq 0$ if and only if $j = n_i$ for some i .
- $S_j(f, g) \neq 0$ if and only if $j = n_i$ or $j = n_i - 1$ for some i .

Normally, $n_{i-1} - n_i = 1$ for $1 < i \leq k$. If $n_{i-1} - n_i > 1$ for some i (the *abnormal* case), then $S_{n_{i-1}-1}(f, g)$ and $S_{n_i}(f, g)$ are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the *subresultant remainder sequence* by setting

$$f_i = S_{n_{i-1}-1}(f, g) \quad (2 \leq i \leq k).$$

In the normal case, these are the same as the $S_{n_i}(f, g)$. He also derived expressions for the constants γ_i in the remainder formulas

$$\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})$$

in terms of the leading coefficients of f_1, \dots, f_{i-1} , working in the field K .

Brown and Traub [BT71] later developed a recursive procedure for computing the coefficients γ_i . Their algorithm deals with elements of the domain A exclusively (assuming $f, g \in A[X]$). However, in the abnormal case there was a problem, a division in A which could only be conjectured to be exact.

This was subsequently justified by Brown [Bro78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of f_i is $s_{n_i-2}(f, g)$ (Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but $s_{n_k}(f, g)$ which is not needed after finding $f_{k+1} = 0$. Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

Polynomial factorization in characteristic zero: Polynomial factorization routines in characteristic zero.

5.1.3 Gröbner basis algorithms

Gröbner bases can be used to answer many problems in computational commutative algebra. Their computation is rather complicated, and very performance-sensitive. We present here various low-level implementations of Gröbner basis computation algorithms; please see the previous section of the manual for usage. Gröbner bases algorithms.

`diofant.polys.groebnertools.buchberger(f, ring)`

Computes Gröbner basis for a set of polynomials in $K[X]$.

Given a set of multivariate polynomials F , finds another set G , such that $\text{Ideal } F = \text{Ideal } G$ and G is a reduced Gröbner basis.

The resulting basis is unique and has monic generators if the ground domains is a field. Otherwise the result is non-unique but Gröbner bases over e.g. integers can be computed (if the input polynomials are monic).

Gröbner bases can be used to choose specific generators for a polynomial ideal. Because these bases are unique you can check for ideal equality by comparing the Gröbner bases. To see if one polynomial lies in an ideal, divide by the elements in the base and see if the remainder vanishes.

They can also be used to solve systems of polynomial equations as, by choosing lexicographic ordering, you can eliminate one variable at a time, provided that the ideal is zero-dimensional (finite number of solutions).

References

- [NKBG03]
- [GMN+91]
- [ALW95]
- [CLOShea15]
- [BW93], page 232

Notes

Used an improved version of Buchberger's algorithm as presented in [BW93].

`diofant.polys.groebnertools.cp_key(c, ring)`

Key for comparing critical pairs.

`diofant.polys.groebnertools.critical_pair(f, g, ring)`

Compute the critical pair corresponding to two labeled polynomials.

A critical pair is a tuple (um, f, vm, g) , where um and vm are terms such that $um * f - vm * g$ is the S-polynomial of f and g (so, wlog assume $um * f > vm * g$). For performance sake, a critical pair is represented as a tuple $(\text{Sign}(um * f), um, f, \text{Sign}(vm * g), vm, g)$, since $um * f$ creates a new, relatively expensive object in memory, whereas $\text{Sign}(um * f)$ and um are lightweight and f (in the tuple) is a reference to an already existing object in memory.

`diofant.polys.groebnertools.f5_reduce(f, B)`

F5-reduce a labeled polynomial f by B .

Continuously searches for non-zero labeled polynomial h in B , such that the leading term lt_h of h divides the leading term lt_f of f and $\text{Sign}(lt_h * h) < \text{Sign}(f)$. If such a labeled polynomial h is found, f gets replaced by $f - lt_f / lt_h * h$. If no such h can be found or f is 0, f is no further F5-reducible and f gets returned.

A polynomial that is reducible in the usual sense need not be F5-reducible, e.g.:

```
>>> _, x, y, z = ring('x y z', QQ, lex)
```

```
>>> f = lbp(sig(Monomial((1, 1, 1)), 4), x, 3)
>>> g = lbp(sig(Monomial((0, 0, 0)), 2), x, 2)
```

```
>>> Polyn(f).div([Polyn(g)])[1]
0
>>> f5_reduce(f, [g])
(((1, 1, 1), 4), x, 3)
```

`diofant.polys.groebnertools.f5b(F, ring)`

Computes a reduced Gröbner basis for the ideal generated by F .

`f5b` is an implementation of the F5B algorithm by Yao Sun and Dingkan Wang. Similarly to Buchberger's algorithm, the algorithm proceeds by computing critical pairs, computing the S-polynomial, reducing it and adjoining the reduced S-polynomial if it is not 0.

Unlike Buchberger's algorithm, each polynomial contains additional information, namely a signature and a number. The signature specifies the path of computation (i.e. from which polynomial in the original basis was it derived and how), the number says when the polynomial was added to the basis. With this information it is (often) possible to decide if an S-polynomial will reduce to 0 and can be discarded.

Optimizations include: Reducing the generators before computing a Gröbner basis, removing redundant critical pairs when a new polynomial enters the basis and sorting the critical pairs and the current basis.

Once a Gröbner basis has been found, it gets reduced.

References

- [SW10]
- [BW93], pp. 203, 216.

`diofant.polys.groebnertools.groebner(seq, ring, method=None)`

Computes Gröbner basis for a set of polynomials in $K[X]$.

Wrapper around the (default) improved Buchberger and the other algorithms for computing Gröbner bases. The choice of algorithm can be changed via `method` argument, where `method` can be either `buchberger` or `f5b`. Default value is determined by `setup()` (page 41).

`diofant.polys.groebnertools.groebner_gcd(f, g)`

Computes GCD of two polynomials using Gröbner bases.

`diofant.polys.groebnertools.groebner_lcm(f, g)`

Computes LCM of two polynomials using Gröbner bases.

The LCM is computed as the unique generator of the intersection of the two ideals generated by f and g . The approach is to compute a Gröbner basis with respect to lexicographic ordering of $t * f$ and $(1 - t) * g$, where t is an unrelated variable and then filtering out the solution that doesn't contain t .

References

- [CLOShea15]

`diofant.polys.groebnertools.is_groebner(G)`

Check if G is a Gröbner basis.

`diofant.polys.groebnertools.is_minimal(G, ring)`

Checks if G is a minimal Gröbner basis.

`diofant.polys.groebnertools.is_rewritable_or_comparable(sign, num, B)`

Check if a labeled polynomial is redundant by checking if its signature and number imply rewritability or comparability.

$(\text{sign}, \text{num})$ is comparable if there exists a labeled polynomial h in B , such that $\text{sign}[1]$ (the index) is less than $\text{Sign}(h)[1]$ and $\text{sign}[0]$ is divisible by the leading monomial of h .

$(\text{sign}, \text{num})$ is rewritable if there exists a labeled polynomial h in B , such that $\text{sign}[1]$ is equal to $\text{Sign}(h)[1]$, $\text{num} < \text{Num}(h)$ and $\text{sign}[0]$ is divisible by $\text{Sign}(h)[0]$.

`diofant.polys.groebnertools.lbp_cmp(f, g)`

Compare two labeled polynomials.

$f < g$ iff

- $\text{Sign}(f) < \text{Sign}(g)$

or

- $\text{Sign}(f) == \text{Sign}(g)$ and $\text{Num}(f) > \text{Num}(g)$

$f > g$ otherwise

`diofant.polys.groebnertools.lbp_key(f)`

Key for comparing two labeled polynomials.

`diofant.polys.groebnertools.lbp_mul_term(f, cx)`

Multiply a labeled polynomial with a term.

The product of a labeled polynomial (s, p, k) by a monomial is defined as $(m * s, m * p, k)$.

`diofant.polys.groebnertools.lbp_sub(f, g)`

Subtract labeled polynomial g from f .

The signature and number of the difference of f and g are signature and number of the maximum of f and g , w.r.t. `lbp_cmp`.

`diofant.polys.groebnertools.matrix_fgfm(F, ring, O_to)`

Converts the reduced Gröbner basis F of a zero-dimensional ideal w.r.t. `O_from` to a reduced Gröbner basis w.r.t. `O_to`.

References

- [\[FaugereGLM93\]](#)

`diofant.polys.groebnertools.red_groebner(G, ring)`

Compute reduced Gröbner basis.

Selects a subset of generators, that already generate the ideal and computes a reduced Gröbner basis for them.

References

- [\[BW93\]](#), page 216.

`diofant.polys.groebnertools.s_poly(cp)`

Compute the S-polynomial of a critical pair.

The S-polynomial of a critical pair cp is $cp[1] * cp[2] - cp[4] * cp[5]$.

`diofant.polys.groebnertools.sig_cmp(u, v, order)`

Compare two signatures by extending the term order to $K[X]^n$.

$u < v$ iff

- the index of v is greater than the index of u

or

- the index of v is equal to the index of u and $u[0] < v[0]$ w.r.t. `order`

$u > v$ otherwise

`diofant.polys.groebnertools.sig_key(s, order)`

Key for comparing two signatures.

$s = (m, k), t = (n, l)$

$s < t$ iff $[k > l]$ or $[k == l \text{ and } m < n]$ $s > t$ otherwise

`diofant.polys.groebnertools.sig_mult(s, m)`

Multiply a signature by a monomial.

The product of a signature (m, i) and a monomial n is defined as (m * t, i).

`diofant.polys.groebnertools.spoly(p1, p2)`

Compute $\text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p1)*p1 - \text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p2)*p2$.

This is the S-poly, provided p1 and p2 are monic

5.1.4 Algebraic number fields

`diofant.polys.numberfields.minpoly_groebner(ex, x, domain)`

Computes the minimal polynomial of an algebraic number using Gröbner bases

Examples

```
>>> minimal_polynomial(sqrt(2) + 1, method='groebner')(x)
x**2 - 2*x - 1
```

References

- [AL94]

5.1.5 Factorization over algebraic number fields

`diofant.polys.factorization_alg_field._alpha_to_z(f, ring)`

Change the representation of a polynomial over $\mathbb{Q}(\alpha)$ by replacing the algebraic element α by a new variable z .

Parameters

- **f** (*PolyElement*) – polynomial in $\mathbb{Q}(\alpha)[x_0, \dots, x_n]$
- **ring** (*PolynomialRing*) – the polynomial ring $\mathbb{Q}[x_0, \dots, x_n, z]$

Returns

f_ (*PolyElement*) – polynomial in $\mathbb{Q}[x_0, \dots, x_n, z]$

`diofant.polys.factorization_alg_field._denominator(f)`

Compute the denominator $\text{den}(f)$ of a polynomial f over \mathbb{Q} , i.e. the smallest integer such that $\text{den}(f)f$ is a polynomial over \mathbb{Z} .

`diofant.polys.factorization_alg_field._diophantine(F, c, A, d, minpoly, p)`

Solve multivariate Diophantine equations over $\mathbb{Z}_p[z]/(\mu(z))$.

`diofant.polys.factorization_alg_field._diophantine_univariate(F, m, minpoly, p)`

Solve univariate Diophantine equations of the form

$$\sum_{f \in F} \left(h_f(x) \cdot \prod_{g \in F \setminus \{f\}} g(x) \right) = x^m$$

over $\mathbb{Z}_p[z]/(\mu(z))$.

`diofant.polys.factorization_alg_field._distinct_prime_divisors(S, domain)`

Find pairwise coprime divisors of all elements of a given list S of integers. If this fails, None is returned.

References

- [JM09], Algorithm 4

`diofant.polys.factorization_alg_field._extended_euclidean_algorithm(f, g, minpoly, p)`

Extended Euclidean Algorithm for univariate polynomials over $\mathbb{Z}_p[z]/(\mu(z))$.

Returns s, t, h , where h is the GCD of f and g and $sf + tg = h$.

`diofant.polys.factorization_alg_field._factor(f)`

Factor a multivariate polynomial f , which is squarefree and primitive in x_0 , in $\mathbb{Q}(\alpha)[x_0, \dots, x_n]$.

References

- [JM09]

`diofant.polys.factorization_alg_field._hensel_lift(f, H, LC, A, minpoly, p)`

Parallel Hensel lifting algorithm over $\mathbb{Z}_p[z]/(\mu(z))$.

Parameters

- **f** (*PolyElement*) – squarefree polynomial in $\mathbb{Z}[x_0, \dots, x_n, z]$
- **H** (*list of PolyElement objects*) – monic univariate factors of $f(x_0, A)$ in $\mathbb{Z}[x_0, z]$
- **LC** (*list of PolyElement objects*) – true leading coefficients of the irreducible factors of f
- **A** (*list of Integer objects*) – the evaluation point $[a_1, \dots, a_n]$
- **p** (*Integer*) – prime number

Returns

pfactors (*list of PolyElement objects*) – irreducible factors of f modulo p

`diofant.polys.factorization_alg_field._leading_coeffs(f, U, gamma, lc_factors, A, D, denoms, divisors)`

Compute the true leading coefficients in x_0 of the irreducible factors of a polynomial f .

If this fails, None is returned.

Parameters

- **f** (*PolyElement*) – squarefree polynomial in $\mathbb{Z}[x_0, \dots, x_n, z]$
- **U** (*list of PolyElement objects*) – monic univariate factors of $f(x_0, A)$ in $\mathbb{Q}(\alpha)[x_0]$
- **gamma** (*Integer*) – integer content of $\text{lc}_{x_0}(f)$
- **lc_factors** (*list of (PolyElement, Integer) objects*) – factorization of $\text{lc}_{x_0}(f)$ in $\mathbb{Z}[x_1, \dots, x_n, z]$

- **A** (*list of Integer objects*) – the evaluation point $[a_1, \dots, a_n]$
- **D** (*Integer*) – integral multiple of the defect of $\mathbb{Q}(\alpha)$
- **denoms** (*list of Integer objects*) – denominators of $\frac{1}{l(A)}$ for l in `lcfactors`
- **divisors** (*list of Integer objects*) – pairwise coprime divisors of all elements of `denoms`

Returns

- **f** (*PolyElement*) – possibly updated polynomial f
- **lcs** (*list of PolyElement objects*) – true leading coefficients of the irreducible factors of f
- **U_** (*list of PolyElement objects*) – list of possibly updated monic associates of the univariate factors U

References

- [JM09]

`diofant.polys.factorization_alg_field._monic_associate(f, ring)`

Compute the monic associate of a polynomial f over $\mathbb{Q}(\alpha)$, which is defined as

$$\text{den} \left(\frac{1}{\text{lc}(f)} f \right) \cdot \frac{1}{\text{lc}(f)} f.$$

The result is a polynomial in $\mathbb{Z}[x_0, \dots, x_n, z]$.

See also:

[`_denominator`](#) (page 771), [`_alpha_to_z`](#) (page 771)

`diofant.polys.factorization_alg_field._padic_lift(f, pfactors, lcs, B, minpoly, p)`

Lift the factorization of a polynomial over $\mathbb{Z}_p[z]/(\mu(z))$ to a factorization over $\mathbb{Z}_{p^m}[z]/(\mu(z))$, where $p^m \geq B$.

If this fails, `None` is returned.

Parameters

- **f** (*PolyElement*) – squarefree polynomial in $\mathbb{Z}[x_0, \dots, x_n, z]$
- **pfactors** (*list of PolyElement objects*) – irreducible factors of f modulo p
- **lcs** (*list of PolyElement objects*) – true leading coefficients in x_0 of the irreducible factors of f
- **B** (*Integer*) – heuristic numerical bound on the size of the largest integer coefficient in the irreducible factors of f
- **minpoly** (*PolyElement*) – minimal polynomial μ of α over \mathbb{Q}
- **p** (*Integer*) – prime number

Returns

H (*list of PolyElement objects*) – factorization of f modulo p^m , where $p^m \geq B$

References

- [\[JM09\]](#)

`diofant.polys.factorization_alg_field._sqf_p(f, minpoly, p)`

Return True if nonzero f is square-free in $\mathbb{Z}_p[z]/(\mu(z))[x]$.

`diofant.polys.factorization_alg_field._subs_ground(f, A)`

Substitute variables in the coefficients of a polynomial f over a PolynomialRing.

`diofant.polys.factorization_alg_field._test_evaluation_points(f, gamma, lcfactors, A, D)`

Test if an evaluation point is suitable for `_factor`.

If it is not, None is returned.

Parameters

- **f** (*PolyElement*) – squarefree polynomial in $\mathbb{Z}[x_0, \dots, x_n, z]$
- **gamma** (*Integer*) – leading coefficient of f in \mathbb{Z}
- **lcfactors** (*list of (PolyElement, Integer) objects*) – factorization of $\text{lc}_{x_0}(f)$ in $\mathbb{Z}[x_1, \dots, x_n, z]$
- **A** (*list of Integer objects*) – the evaluation point $[a_1, \dots, a_n]$
- **D** (*Integer*) – integral multiple of the defect of $\mathbb{Q}(\alpha)$

Returns

- **fA** (*PolyElement*) – f evaluated at A , i.e. $f(x_0, A)$
- **denoms** (*list of Integer objects*) – the denominators of $\frac{1}{l(A)}$ for l in `lcfactors`
- **divisors** (*list of Integer objects*) – pairwise coprime divisors of all elements of `denoms`

References

- [\[JM09\]](#)

See also:

[_factor](#) (page 772)

`diofant.polys.factorization_alg_field._test_prime(f, A, minpoly, p)`

Test if a prime number is suitable for `_factor`.

See also:

[_factor](#) (page 772)

`diofant.polys.factorization_alg_field._z_to_alpha(f, ring)`

Change the representation of a polynomial in $\mathbb{Q}[x_0, \dots, x_n, z]$ by replacing the variable z by the algebraic element α of the given ring $\mathbb{Q}(\alpha)[x_0, \dots, x_n]$.

Parameters

- **f** (*PolyElement*) – polynomial in $\mathbb{Q}[x_0, \dots, x_n, z]$

- **ring** (*PolynomialRing*) – the polynomial ring $\mathbb{Q}(\alpha)[x_0, \dots, x_n]$

Returns

f_ (*PolyElement*) – polynomial in $\mathbb{Q}(\alpha)[x_0, \dots, x_n]$

`diofant.polys.factorization_alg_field.efactor(f)`

Factor multivariate polynomial f over algebraic number fields.

References

- [JM09]

`diofant.polys.factorization_alg_field.trager(f)`

Factor multivariate polynomial f over algebraic number fields, using classical Trager algorithm.

References

- [Tra76]

5.1.6 Modular GCD

`diofant.polys.modulargcd._chinese_remainder_reconstruction(hp, hq, p, q)`

Construct a polynomial h_{pq} in $\mathbb{Z}_{pq}[x_0, \dots, x_{k-1}]$ such that

$$h_{pq} = h_p \bmod p$$

$$h_{pq} = h_q \bmod q$$

for relatively prime integers p and q and polynomials h_p and h_q in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ and $\mathbb{Z}_q[x_0, \dots, x_{k-1}]$ respectively.

The coefficients of the polynomial h_{pq} are computed with the Chinese Remainder Theorem. The symmetric representation in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$, $\mathbb{Z}_q[x_0, \dots, x_{k-1}]$ and $\mathbb{Z}_{pq}[x_0, \dots, x_{k-1}]$ is used.

Parameters

- **hp** (*PolyElement*) – multivariate integer polynomial with coefficients in \mathbb{Z}_p
- **hq** (*PolyElement*) – multivariate integer polynomial with coefficients in \mathbb{Z}_q
- **p** (*Integer*) – modulus of h_p , relatively prime to q
- **q** (*Integer*) – modulus of h_q , relatively prime to p

Examples

```
>>> _, x, y = ring('x y', ZZ)
>>> p = 3
>>> q = 5
```

```
>>> hp = x**3*y - x**2 - 1
>>> hq = -x**3*y - 2*x*y**2 + 2
```

```
>>> hpq = _chinese_remainder_reconstruction(hp, hq, p, q)
>>> hpq
4*x**3*y + 5*x**2 + 3*x*y**2 + 2
```

```
>>> hpq.trunc_ground(p) == hp
True
>>> hpq.trunc_ground(q) == hq
True
```

```
>>> _, x, y, z = ring('x y z', ZZ)
>>> p = 6
>>> q = 5
```

```
>>> hp = 3*x**4 - y**3*z + z
>>> hq = -2*x**4 + z
```

```
>>> hpq = _chinese_remainder_reconstruction(hp, hq, p, q)
>>> hpq
3*x**4 + 5*y**3*z + z
```

```
>>> hpq.trunc_ground(p) == hp
True
>>> hpq.trunc_ground(q) == hq
True
```

`diofant.polys.modulargcd._div(f, g, minpoly, p)`

Division with remainder for univariate polynomials over $\mathbb{Z}_p[z]/(\mu(z))$.

`diofant.polys.modulargcd._euclidean_algorithm(f, g, minpoly, p)`

Compute the monic GCD of two univariate polynomials in $\mathbb{Z}_p[z]/(\tilde{m}_\alpha(z))[x]$ with the Euclidean Algorithm.

In general, $\tilde{m}_\alpha(z)$ is not irreducible, so it is possible that some leading coefficient is not invertible modulo $\tilde{m}_\alpha(z)$. In that case `None` is returned.

Parameters

- **f, g** (*PolyElement*) – polynomials in $\mathbb{Z}[x, z]$
- **minpoly** (*PolyElement*) – polynomial in $\mathbb{Z}[z]$, not necessarily irreducible
- **p** (*Integer*) – prime number, modulus of \mathbb{Z}_p

Returns

h (*PolyElement*) – GCD of f and g in $\mathbb{Z}[z, x]$ or `None`, coefficients are in $[-\frac{p-1}{2}, \frac{p-1}{2}]$

`diofant.polys.modulargcd._evaluate_ground(f, i, a)`

Evaluate a polynomial f at a in the i -th variable of the ground domain.

`diofant.polys.modulargcd._func_field_modgcd_m(f, g, minpoly)`

Compute the GCD of two polynomials in $\mathbb{Q}(t_1, \dots, t_k)[z]/(m_\alpha(z))[x]$ using a modular algorithm.

The algorithm computes the GCD of two polynomials f and g by calculating the GCD in $\mathbb{Z}_p(t_1, \dots, t_k)[z]/(\tilde{m}_\alpha(z))[x]$ for suitable primes p and the primitive associate $\tilde{m}_\alpha(z)$ of $m_\alpha(z)$. Then the coefficients are reconstructed with the Chinese Remainder Theorem and Rational Reconstruction. To compute the GCD over $\mathbb{Z}_p(t_1, \dots, t_k)[z]/(\tilde{m}_\alpha(z))[x]$, the recursive subroutine `_func_field_modgcd_p` is used. To verify the result in $\mathbb{Q}(t_1, \dots, t_k)[z]/(m_\alpha(z))[x]$, a fraction free trial division is used.

Parameters

- **f, g** (*PolyElement*) - polynomials in $\mathbb{Z}[t_1, \dots, t_k][x, z]$
- **minpoly** (*PolyElement*) - irreducible polynomial in $\mathbb{Z}[t_1, \dots, t_k][z]$

Returns

h (*PolyElement*) - the primitive associate in $\mathbb{Z}[t_1, \dots, t_k][x, z]$ of the GCD of f and g

Examples

```
>>> _, x, z = ring('x z', ZZ)
>>> minpoly = (z**2 - 2).drop(0)
```

```
>>> f = x**2 + z + 2*x*z + 2
>>> g = x**2 + z + 2*x*z + 2
>>> func_field_modgcd_m(f, g, minpoly)
x + z
```

```
>>> D, t = ring('t', ZZ)
>>> _, x, z = ring('x z', D)
>>> minpoly = (z**2 - 3).drop(0)
```

```
>>> f = x**2 + (t + 1)*x*z + 3*t
>>> g = x**2 + 3*t
>>> func_field_modgcd_m(f, g, minpoly)
x + t*z
```

References

- [MvH04]

See also:

`_func_field_modgcd_p` (page 777)

`diofant.polys.modulargcd._func_field_modgcd_p(f, g, minpoly, p)`

Compute the GCD of two polynomials f and g in $\mathbb{Z}_p(t_1, \dots, t_k)[z]/(\tilde{m}_\alpha(z))[x]$.

The algorithm reduces the problem step by step by evaluating the polynomials f and g at $t_k = a$ for suitable $a \in \mathbb{Z}_p$ and then calls itself recursively to compute the GCD in $\mathbb{Z}_p(t_1, \dots, t_{k-1})[z]/(\tilde{m}_\alpha(z))[x]$. If these recursive calls are successful, the GCD over k variables is interpolated, otherwise the algorithm returns None. After interpolation, Rational Function Reconstruction is used to obtain the correct coefficients. If this fails, a new evaluation point has to be chosen, otherwise the desired polynomial is obtained by clearing denominators. The result is verified with a fraction free trial division.

Parameters

- **f, g** (*PolyElement*) - polynomials in $\mathbb{Z}[t_1, \dots, t_k][x, z]$

- **minpoly** (*PolyElement*) – polynomial in $\mathbb{Z}[t_1, \dots, t_k][z]$, not necessarily irreducible
- **p** (*Integer*) – prime number, modulus of \mathbb{Z}_p

Returns

h (*PolyElement*) – primitive associate in $\mathbb{Z}[t_1, \dots, t_k][x, z]$ of the GCD of the polynomials f and g or `None`, coefficients are in $[-\frac{p-1}{2}, \frac{p-1}{2}]$

References

- [MvH04]

`diofant.polys.modulargcd._gf_gcdex(f, g, p)`

Extended Euclidean Algorithm for two univariate polynomials over \mathbb{Z}_p .

Returns polynomials s, t and h , such that h is the GCD of f and g and $sf + tg = h \bmod p$.

`diofant.polys.modulargcd._interpolate(evalpoints, hpeval, ring, i, p, ground=False)`

Reconstruct a polynomial h_p in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ from a list of evaluation points in \mathbb{Z}_p and a list of polynomials in $\mathbb{Z}_p[x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{k-1}]$, which are the images of h_p evaluated in the variable x_i .

It is also possible to reconstruct a parameter of the ground domain, i.e. if h_p is a polynomial over $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$. In this case, one has to set `ground=True`.

Parameters

- **evalpoints** (*list of Integer objects*) – list of evaluation points in \mathbb{Z}_p
- **hpeval** (*list of PolyElement objects*) – list of polynomials in (resp. over) $\mathbb{Z}_p[x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{k-1}]$, images of h_p evaluated in the variable x_i
- **ring** (*PolynomialRing*) – h_p will be an element of this ring
- **i** (*Integer*) – index of the variable which has to be reconstructed
- **p** (*Integer*) – prime number, modulus of h_p
- **ground** (*Boolean*) – indicates whether x_i is in the ground domain, default is `False`

Returns

hp (*PolyElement*) – interpolated polynomial in (resp. over) $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$

`diofant.polys.modulargcd._minpoly_from_dense(minpoly, ring)`

Change representation of the minimal polynomial from `Poly` to `PolyElement` for a given ring.

`diofant.polys.modulargcd._modgcd_p(f, g, degbound, contbound)`

Compute the GCD of two polynomials in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$.

The algorithm reduces the problem step by step by evaluating the polynomials f and g at $x_{k-1} = a$ for suitable $a \in \mathbb{Z}_p$ and then calls itself recursively to compute the GCD in $\mathbb{Z}_p[x_0, \dots, x_{k-2}]$. If these recursive calls are successful for enough evaluation points, the GCD in k variables is interpolated, otherwise the algorithm returns `None`. Every time a GCD or a content is computed, their degrees are compared with the bounds. If a degree greater than the bound is encountered, then the current call returns `None` and a new evaluation point has to be chosen. If at some point the degree is smaller, the correspondent bound is updated and the algorithm fails.

Parameters

- **f** (*PolyElement*) – multivariate polynomial with coefficients in \mathbb{Z}_p
- **g** (*PolyElement*) – multivariate polynomial with coefficients in \mathbb{Z}_p
- **degbound** (*list of Integer objects*) – `degbound[i]` is an upper bound for the degree of the GCD of f and g in the variable x_i
- **contbound** (*list of Integer objects*) – `contbound[i]` is an upper bound for the degree of the content of the GCD in $\mathbb{Z}_p[x_i][x_0, \dots, x_{i-1}]$, `contbound[0]` is not used can therefore be chosen arbitrarily.

Returns

h (*PolyElement*) – GCD of the polynomials f and g or None

References

- [MW00]
- [Bro71]

`diofant.polys.modulargcd._primitive_in_x0(f)`

Compute the content in x_0 and the primitive part of a polynomial f in $\mathbb{Q}(\alpha)[x_0, x_1, \dots, x_{n-1}] \cong \mathbb{Q}(\alpha)[x_1, \dots, x_{n-1}][x_0]$.

`diofant.polys.modulargcd._rational_function_reconstruction(c, p, m)`

Reconstruct a rational function $\frac{a}{b}$ in $\mathbb{Z}_p(t)$ from

$$c = \frac{a}{b} \bmod m,$$

where c and m are polynomials in $\mathbb{Z}_p[t]$ and m has positive degree.

The algorithm is based on the Euclidean Algorithm. In general, m is not irreducible, so it is possible that b is not invertible modulo m . In that case None is returned.

Parameters

- **c** (*PolyElement*) – univariate polynomial in $\mathbb{Z}[t]$
- **p** (*Integer*) – prime number
- **m** (*PolyElement*) – modulus, not necessarily irreducible

Returns

frac (*FracElement*) – either $\frac{a}{b}$ in $\mathbb{Z}(t)$ or None

References

- [MvH04]

`diofant.polys.modulargcd._rational_reconstruction_func_coeffs(hm, p, m, ring, k)`

Reconstruct every coefficient c_h of a polynomial h in $\mathbb{Z}_p(t_k)[t_1, \dots, t_{k-1}][x, z]$ from the corresponding coefficient c_{h_m} of a polynomial h_m in $\mathbb{Z}_p[t_1, \dots, t_k][x, z] \cong \mathbb{Z}_p[t_k][t_1, \dots, t_{k-1}][x, z]$ such that

$$c_{h_m} = c_h \bmod m,$$

where $m \in \mathbb{Z}_p[t]$.

The reconstruction is based on the Euclidean Algorithm. In general, m is not irreducible, so it is possible that this fails for some coefficient. In that case `None` is returned.

Parameters

- **hm** (*PolyElement*) – polynomial in $\mathbb{Z}[t_1, \dots, t_k][x, z]$
- **p** (*Integer*) – prime number, modulus of \mathbb{Z}_p
- **m** (*PolyElement*) – modulus, polynomial in $\mathbb{Z}[t]$, not necessarily irreducible
- **ring** (*PolynomialRing*) – $\mathbb{Z}(t_k)[t_1, \dots, t_{k-1}][x, z]$, h will be an element of this ring
- **k** (*Integer*) – index of the parameter t_k which will be reconstructed

Returns

h (*PolyElement*) – reconstructed polynomial in $\mathbb{Z}(t_k)[t_1, \dots, t_{k-1}][x, z]$ or `None`

See also:

[_rational_function_reconstruction](#) (page 779)

`diofant.polys.modulargcd._rational_reconstruction_int_coeffs(hm, m, ring)`

Reconstruct every rational coefficient c_h of a polynomial h in $\mathbb{Q}[t_1, \dots, t_k][x, z]$ from the corresponding integer coefficient c_{h_m} of a polynomial h_m in $\mathbb{Z}[t_1, \dots, t_k][x, z]$ such that

$$c_{h_m} = c_h \bmod m,$$

where $m \in \mathbb{Z}$.

The reconstruction is based on the Euclidean Algorithm. In general, m is not a prime number, so it is possible that this fails for some coefficient. In that case `None` is returned.

Parameters

- **hm** (*PolyElement*) – polynomial in $\mathbb{Z}[t_1, \dots, t_k][x, z]$
- **m** (*Integer*) – modulus, not necessarily prime
- **ring** (*PolynomialRing*) – $\mathbb{Q}[t_1, \dots, t_k][x, z]$, h will be an element of this ring

Returns

h (*PolyElement*) – reconstructed polynomial in $\mathbb{Q}[t_1, \dots, t_k][x, z]$ or `None`

See also:

[diofant.ntheory.modular.integer_rational_reconstruction](#) (page 247)

`diofant.polys.modulargcd._to_ANP_poly(f, ring)`

Convert a polynomial $f \in \mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\tilde{m}_\alpha(z))[x_0]$ to a polynomial in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$, where $\tilde{m}_\alpha(z) \in \mathbb{Z}[z]$ is the primitive associate of the minimal polynomial $m_\alpha(z)$ of α over \mathbb{Q} .

Parameters

- **f** (*PolyElement*) – polynomial in $\mathbb{Z}[x_1, \dots, x_{n-1}][x_0, z]$
- **ring** (*PolynomialRing*) – $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

Returns

f_ (*PolyElement*) – polynomial in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

`diofant.polys.modulargcd._to_ZZ_poly(f, ring)`

Compute an associate of a polynomial $f \in \mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$ in $\mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\check{m}_\alpha(z))[x_0]$, where $\check{m}_\alpha(z) \in \mathbb{Z}[z]$ is the primitive associate of the minimal polynomial $m_\alpha(z)$ of α over \mathbb{Q} .

Parameters

- **f** (*PolyElement*) – polynomial in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$
- **ring** (*PolynomialRing*) – $\mathbb{Z}[x_1, \dots, x_{n-1}][x_0, z]$

Returns

f_ (*PolyElement*) – associate of f in $\mathbb{Z}[x_1, \dots, x_{n-1}][x_0, z]$

`diofant.polys.modulargcd._trunc(f, minpoly, p)`

Compute the reduced representation of a polynomial f in $\mathbb{Z}_p[z]/(\check{m}_\alpha(z))[x]$

Parameters

- **f** (*PolyElement*) – polynomial in $\mathbb{Z}[x, z]$
- **minpoly** (*PolyElement*) – polynomial $\check{m}_\alpha \in \mathbb{Z}[z]$, not necessarily irreducible
- **p** (*Integer*) – prime number, modulus of \mathbb{Z}_p

Returns

ftrunc (*PolyElement*) – polynomial in $\mathbb{Z}[x, z]$, reduced modulo $\check{m}_\alpha(z)$ and p

`diofant.polys.modulargcd.func_field_modgcd(f, g)`

Compute the GCD of two polynomials f and g in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$ using a modular algorithm.

The algorithm first computes the primitive associate $\check{m}_\alpha(z)$ of the minimal polynomial m_α in $\mathbb{Z}[z]$ and the primitive associates of f and g in $\mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\check{m}_\alpha)[x_0]$. Then it computes the GCD in $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$. This is done by calculating the GCD in $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstruction. The GCD over $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ is computed with a recursive subroutine, which evaluates the polynomials at $x_{n-1} = a$ for suitable evaluation points $a \in \mathbb{Z}_p$ and then calls itself recursively until the ground domain does no longer contain any parameters. For $\mathbb{Z}_p[z]/(\check{m}_\alpha(z))[x_0]$ the Euclidean Algorithm is used. The results of those recursive calls are then interpolated and Rational Function Reconstruction is used to obtain the correct coefficients. The results, both in $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$ and $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$, are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in $\mathbb{Q}(\alpha)[x_1, \dots, x_{n-1}]$ have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this `func_field_modgcd` is called recursively.

Parameters

f, g (*PolyElement*) – polynomials in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

Returns

h (*PolyElement*) – monic GCD of the polynomials f and g

Examples

```
>>> A = QQ.algebraic_field(sqrt(2))
>>> _, x = ring('x', A)
```

```
>>> func_field_modgcd(x**2 - 2, x + sqrt(2))
x + sqrt(2)
```

```
>>> _, x, y = ring('x y', A)
```

```
>>> func_field_modgcd((x + sqrt(2)*y)**2, x + sqrt(2)*y)
x + sqrt(2)*y
```

```
>>> func_field_modgcd(x + sqrt(2)*y, x + y)
1
```

References

- [\[MvH04\]](#)

`diofant.polys.modulargcd.modgcd(f, g)`

Compute the GCD of two polynomials in $\mathbb{Z}[x_0, \dots, x_{k-1}]$ using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over \mathbb{Z}_p the recursive subroutine `_modgcd_p` is used. To verify the result in $\mathbb{Z}[x_0, \dots, x_{k-1}]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters

- **f** (*PolyElement*) - multivariate integer polynomial
- **g** (*PolyElement*) - multivariate integer polynomial

Returns

h (*PolyElement*) - GCD of the polynomials f and g

Examples

```
>>> _, x, y = ring('x y', ZZ)
```

```
>>> modgcd((x - y)*(x + y), (x + y)**2)
x + y
```

```
>>> _, x, y, z = ring('x y z', ZZ)
```

```
>>> modgcd((x - y)*z**2, (x**2 + 1)*z)
z
```

References

- [\[MW00\]](#)
- [\[Bro71\]](#)

`diofant.polys.modulargcd.trial_division(f, h, minpoly, p=None)`

Check if h divides f in $\mathbb{K}[t_1, \dots, t_k][z]/(m_\alpha(z))$, where \mathbb{K} is either \mathbb{Q} or \mathbb{Z}_p .

This algorithm is based on pseudo division and does not use any fractions. By default \mathbb{K} is \mathbb{Q} , if a prime number p is given, \mathbb{Z}_p is chosen instead.

Parameters

- **f, h** (*PolyElement*) – polynomials in $\mathbb{Z}[t_1, \dots, t_k][x, z]$
- **minpoly** (*PolyElement*) – polynomial $m_\alpha(z)$ in $\mathbb{Z}[t_1, \dots, t_k][z]$
- **p** (*Integer or None*) – if p is given, \mathbb{K} is set to \mathbb{Z}_p instead of \mathbb{Q} , default is `None`

Returns

rem (*PolyElement*) – remainder of $\frac{f}{h}$

References

- [\[vHM02\]](#)

5.1.7 Heuristic GCD

`_GCD._zz_heu_gcd(f, g)`

Heuristic polynomial GCD in $\mathbb{Z}[X]$.

Given univariate polynomials f and g in $\mathbb{Z}[X]$, returns their GCD, i.e. polynomial h :

`h = gcd(f, g)`

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.

References

- [LF95]

5.1.8 Further tools

Real and complex root isolation and refinement algorithms.

class diofant.polys.rootisolation.**ComplexInterval**(*a, b, I, Q, F1, F2, f1, f2, conj=False*)

A fully qualified representation of a complex isolation interval. The printed form is shown as (x1, y1) x (x2, y2): the southwest x northeast coordinates of the interval's rectangle.

as_tuple()

Return tuple representation of complex isolating interval.

property ax

Return x coordinate of south-western corner.

property ay

Return y coordinate of south-western corner.

property bx

Return x coordinate of north-eastern corner.

property by

Return y coordinate of north-eastern corner.

property center

Return the center of the complex isolating interval.

conjugate()

Return conjugated isolating interval.

is_disjoint(*other, check_re_refinement=False, re_disjoint=False*)

Return True if two isolation intervals are disjoint.

Parameters

- **check_re_refinement** (*bool, optional*) - If enabled, test that either real projections of isolation intervals are disjoint or roots share common real part.
- **re_disjoint** (*bool, optional*) - If enabled, return True only if real projections of isolation intervals are disjoint.

refine(*vertical=False*)

Perform one step of complex root refinement algorithm.

class diofant.polys.rootisolation.**RealInterval**(*data, f*)

A fully qualified representation of a real isolation interval.

property a

Return the position of the left end.

as_tuple()

Return tuple representation of real isolating interval.

property b

Return the position of the right end.

property center

Return the center of the real isolating interval.

is_disjoint(*other*)

Return True if two isolation intervals are disjoint.

refine()

Perform one step of real root refinement algorithm.

Square-free decomposition algorithms and related tools.

5.1.9 Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute! Options manager for *Poly* (page 507) and public API functions.

class diofant.polys.polyoptions.Order

order option to polynomial manipulation functions.

Definitions of common exceptions for *polys* (page 505) module.

exception diofant.polys.polyerrors.BasePolynomialError

Base class for polynomial related exceptions.

exception diofant.polys.polyerrors.CoercionFailedError

Raised when a coercion is failed.

exception diofant.polys.polyerrors.ComputationFailedError(*func, nargs, exc*)

Raised when polynomial computation failed.

exception diofant.polys.polyerrors.DomainError

Generic domain error.

exception diofant.polys.polyerrors.EvaluationFailedError

Raised when a polynomial evaluation is failed.

exception diofant.polys.polyerrors.ExactQuotientFailedError(*f, g, dom=None*)

Raised when exact quotient is failed.

exception diofant.polys.polyerrors.ExtraneousFactorsError

Raised when there are extraneous factors.

exception diofant.polys.polyerrors.FlagError

Generic flag error.

exception diofant.polys.polyerrors.GeneratorsError

Raised when polynomial generators are unsuitable.

exception diofant.polys.polyerrors.GeneratorsNeededError

Raised when more generators needed.

exception diofant.polys.polyerrors.HeuristicGCDFailedError

Raised when a heuristic GCD is failed.

exception diofant.polys.polyerrors.**HomomorphismFailedError**
Raised when a homomorphism is failed.

exception diofant.polys.polyerrors.**IsomorphismFailedError**
Raised when an isomorphism is failed.

exception diofant.polys.polyerrors.**ModularGCDFailedError**
Raised when a modular GCD is failed.

exception diofant.polys.polyerrors.**MultivariatePolynomialError**
Generic multivariate polynomial error.

exception diofant.polys.polyerrors.**NotAlgebraicError**
Raised when a non algebraic element occurred.

exception diofant.polys.polyerrors.**NotInvertibleError**
Raised when a element is not invertible.

exception diofant.polys.polyerrors.**NotReversibleError**
Raised when a element is not reversible.

exception diofant.polys.polyerrors.**OperationNotSupportedError**(*poly, func*)
Raised when an operation is not supported.

exception diofant.polys.polyerrors.**OptionError**
Generic option error.

exception diofant.polys.polyerrors.**PolificationFailedError**(*opt, origs, exprs, seq=False*)
Raised if polunomial construction is failed.

exception diofant.polys.polyerrors.**PolynomialDivisionFailedError**(*f, g, domain*)
Raised when polynomial division is failed.

exception diofant.polys.polyerrors.**PolynomialError**
Generic polynomial error.

exception diofant.polys.polyerrors.**RefinementFailedError**
Raised when a root refinement is failed.

exception diofant.polys.polyerrors.**UnificationFailedError**
Raised when domains unification failed.

exception diofant.polys.polyerrors.**UnivariatePolynomialError**
Generic univariate polynomial error.

exception diofant.polys.polyerrors.**UnluckyLeadingCoefficientError**
Raised when there are unlucky LC.

class diofant.polys.fields.**FracElement**(*numer, denom=None*)
Element of multivariate distributed rational function field.

See also:
[*FractionField*](#) (page 431)

compose(*x, a=None*)
Computes the functional composition.

diff(*x*)
Computes partial derivative in x.

Examples

```
>>> x, y, z = field('x y z', ZZ)
>>> ((x**2 + y)/(z + 1)).diff(x)
2*x/(z + 1)
```

5.2 The Gruntz Algorithm

This section explains the basics of the algorithm [Gru96] used for computing limits. Most of the time the `limit()` (page 753) 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 of single variable x according to how rapidly varying they at infinity. Any two functions $f(x)$ and $g(x)$ can be compared using the properties of:

$$L = \lim_{x \rightarrow \infty} \frac{\log |f(x)|}{\log |g(x)|}$$

We shall say that $f(x)$ *dominates* $g(x)$, written $f(x) \succ g(x)$, iff $L = \pm\infty$. 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 $f(x) \asymp g(x)$.

It is easy to show the following examples:

- $e^{e^x} \succ e^{x^2} \succ e^x \succ x \succ 42$
- $2 \asymp 3 \asymp -5$
- $x \asymp x^2 \asymp x^3 \asymp -x$
- $e^x \asymp e^{-x} \asymp e^{2x} \asymp e^{x+e^{-x}}$
- $f(x) \asymp 1/f(x)$

Using these definitions yields the following strategy for computing $\lim_{x \rightarrow \infty} f(x)$:

1. Given the function $f(x)$, we find the set of *most rapidly varying subexpressions* (MRV set) of it. All items of this set belongs to the same comparability class. Let's say it is $\{e^x, e^{2x}\}$.
2. Choose an expression ω which is positive and tends to zero and which is in the same comparability class as any element of the MRV set. Such element always exists. Then we rewrite the MRV set using ω , in our case $\{\omega^{-1}, \omega^{-2}\}$, and substitute it into $f(x)$.
3. Let $f(\omega)$ be the function which is obtained from $f(x)$ after the rewrite step above. Consider all expressions independent of ω as constants and compute the leading term of the power series of $f(\omega)$ around $\omega = 0^+$:

$$f(\omega) = c_0 \omega^{e_0} + c_1 \omega^{e_1} + \dots$$

where $e_0 < e_1 < e_2 \dots$

4. If the leading exponent $e_0 > 0$ then the limit is 0. If $e_0 < 0$, then the answer is $\pm\infty$ (depends on sign of c_0). Finally, if $e_0 = 0$, the limit is the limit of the leading coefficient c_0 .

Notes

This exposition glossed over several details. For example, limits could be computed recursively (steps 1 and 4). Please address to the Gruntz thesis [Gru96] for proof of the termination (pp. 52-60).

`diofant.calculus.gruntz.leadterm(e, x)`

Compute the leading term of the series.

Returns

tuple - The leading term $c_0 w^{e_0}$ of the series of *e* in terms of the most rapidly varying subexpression *w* in form of the pair (*c0*, *e0*) of Expr.

Examples

```
>>> leadterm(1/exp(-x + exp(-x)) - exp(x), x)
(-1, 0)
```

`diofant.calculus.gruntz.limitinf(e, x)`

Compute the limit of the expression at the infinity.

Examples

```
>>> limitinf(exp(x)*(exp(1/x - exp(-x)) - exp(1/x)), x)
-1
```

`diofant.calculus.gruntz.mrv(e, x)`

Calculate the MRV set of the expression.

Examples

```
>>> mrv(log(x - log(x))/log(x), x)
{x}
```

`diofant.calculus.gruntz.mrv_max(f, g, x)`

Compute the maximum of two MRV sets.

Examples

```
>>> mrv_max({log(x)}, {x**5}, x)
{x**5}
```

`diofant.calculus.gruntz.rewrite(e, x, w)`

Rewrites the expression in terms of the MRV subexpression.

Parameters

- **e** (*Expr*) - an expression
- **x** (*Symbol*) - variable of the *e*
- **w** (*Symbol*) - The symbol which is going to be used for substitution in place of the MRV in *x* subexpression.

Returns

tuple – A pair: rewritten (in w) expression and $\log(w)$.

Examples

```
>>> rewrite(exp(x)*log(x), x, y)
(log(x)/y, -x)
```

```
diofant.calculus.gruntz.signinf(e, x)
```

Determine sign of the expression at the infinity.

Returns

$\{1, 0, -1\}$ – One or minus one, if $e > 0$ or $e < 0$ for x sufficiently large and zero if e is *constantly* zero for $x \rightarrow \infty$.

5.3 Details on the Hypergeometric Function Expansion

This page describes how the function `hyperexpand()` (page 603) and related code work. For usage, see the documentation of the `simplify` module.

5.3.1 Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roa96] and [Roa97].

Recall that the hypergeometric function is (initially) defined as

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n}{(b_1)_n \dots (b_q)_n} \frac{z^n}{n!}.$$

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 conjunction with a “for all i ”. E.g. $a_i < 4$ for all i . Uppercase subscripts I and J denote a chosen, fixed index. So for example $a_I > 0$ is true if the inequality holds for the one index I we are currently interested in.

Incrementing and decrementing indices

Suppose $a_i \neq 0$. Set $A(a_i) = \frac{z}{a_i} \frac{d}{dz} + 1$. It is then easy to show that $A(a_i) {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_pF_q \left(\begin{smallmatrix} a_p + e_i \\ b_q \end{smallmatrix} \middle| z \right)$, where e_i is the i -th unit vector. Similarly for $b_j \neq 1$ we set $B(b_j) = \frac{z}{b_j - 1} \frac{d}{dz} + 1$ and find $B(b_j) {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q - e_j \end{smallmatrix} \middle| z \right)$. Thus we can increment upper and decrement lower indices at will, as long as we don't go through zero. The $A(a_i)$ and $B(b_j)$ are called shift operators.

It is also easy to show that $\frac{d}{dz} {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = \frac{a_1 \dots a_p}{b_1 \dots b_q} {}_pF_q \left(\begin{smallmatrix} a_p + 1 \\ b_q + 1 \end{smallmatrix} \middle| z \right)$, where $a_p + 1$ is the vector $a_1 + 1, a_2 + 1, \dots$ and similarly for $b_q + 1$. Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: $\left[\frac{d}{dz} \prod_{j=1}^q B(b_j) - \frac{a_1 \dots a_p}{(b_1 - 1) \dots (b_q - 1)} \prod_{i=1}^p A(a_i) \right] {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$. This holds if all shift operators are defined, i.e. if no $a_i = 0$ and no $b_j = 1$. Clearing denominators and multiplying through by z we arrive at the following equation: $\left[z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) - z \prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) \right] {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$. Even though our derivation does not show it, it can be checked that this equation holds whenever the ${}_pF_q$ is defined.

Notice that, under suitable conditions on a_I, b_J , each of the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ can be expressed in terms of $A(a_I)$ or $B(b_J)$. Our next aim is to write the Hypergeometric differential equation as follows: $[XA(a_I) - r] {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$, for some operator X and some constant r to be determined. If $r \neq 0$, then we can write this as $\frac{-1}{r} X {}_pF_q \left(\begin{smallmatrix} a_p + e_I \\ b_q \end{smallmatrix} \middle| z \right) = {}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right)$, and so $\frac{-1}{r} X$ undoes the shifting of $A(a_I)$, whence it will be called an inverse-shift operator.

Now $A(a_I)$ exists if $a_I \neq 0$, and then $z \frac{d}{dz} = a_I A(a_I) - a_I$. Observe also that all the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ commute. We have $\prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) = \left(\prod_{i=1, i \neq I}^p \left(z \frac{d}{dz} + a_i \right) \right) a_I A(a_I)$, so this gives us the first half of X . The other half does not have such a nice expression. We find $z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) = (a_I A(a_I) - a_I) \prod_{j=1}^q (a_I A(a_I) - a_I + b_j - 1)$. Since the first half had no constant term, we infer $r = -a_I \prod_{j=1}^q (b_j - 1 - a_I)$.

This tells us under which conditions we can “un-shift” $A(a_I)$, namely when $a_I \neq 0$ and $r \neq 0$. Substituting $a_I - 1$ for a_I then tells us under what conditions we can decrement the index a_I . Doing a similar analysis for $B(b_J)$, we arrive at the following rules:

- An index a_I can be decremented if $a_I \neq 1$ and $a_I \neq b_j$ for all b_j .
- An index b_J can be incremented if $b_J \neq -1$ and $b_J \neq a_i$ for all a_i .

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

Reduction of Order

Notice that, quite trivially, if $a_I = b_J$, we have ${}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_{p-1}F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$, where a_p^* means a_p with a_I omitted, and similarly for b_q^* . We call this reduction of order.

In fact, we can do even better. If $a_I - b_J \in \mathbb{Z}_{>0}$, then it is easy to see that $\frac{(a_I)_n}{(b_J)_n}$ is actually a polynomial in n . It is also easy to see that $(z \frac{d}{dz})^k z^n = n^k z^n$. Combining these two remarks we find:

If $a_I - b_J \in \mathbb{Z}_{>0}$, then there exists a polynomial $p(n) = p_0 + p_1 n + \dots$ (of degree $a_I - b_J$) such that $\frac{(a_I)_n}{(b_J)_n} = p(n)$ and ${}_pF_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = \left(p_0 + p_1 z \frac{d}{dz} + p_2 \left(z \frac{d}{dz} \right)^2 + \dots \right) {}_{p-1}F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$.

Thus any set of parameters a_p, b_q is reachable from a set of parameters c_r, d_s where $c_i - d_j \in \mathbb{Z}$ implies $c_i < d_j$. Such a set of parameters c_r, d_s is called suitable. Our database of known formulae should only contain suitable origins. The reasons are twofold: firstly, working from suitable origins is easier, and secondly, a formula for a non-suitable origin can be deduced from a lower order formula, and we should put this one into the database instead.

Moving Around in the Parameter Space

It remains to investigate the following question: suppose a_p, b_q and a_p^0, b_q^0 are both suitable, and also $a_i - a_i^0 \in \mathbb{Z}$, $b_j - b_j^0 \in \mathbb{Z}$. When is a_p, b_q reachable from a_p^0, b_q^0 ? It is clear that we can treat all parameters independently that are incongruent mod 1. So assume that a_i and b_j are congruent to r mod 1, for all i and j . The same then follows for a_i^0 and b_j^0 .

If $r \neq 0$, then any such a_p, b_q is reachable from any a_p^0, b_q^0 . To see this notice that there exist constants c, c^0 , congruent mod 1, such that $a_i < c < b_j$ for all i and j , and similarly $a_i^0 < c^0 < b_j^0$. If $n = c - c^0 > 0$ then we first inverse-shift all the b_j^0 n times up, and then similarly shift up all the a_i^0 n times. If $n < 0$ then we first inverse-shift down the a_i^0 and then shift down the b_j^0 . This reduces to the case $c = c^0$. But evidently we can now shift or inverse-shift around the a_i^0 arbitrarily so long as we keep them less than c , and similarly for the b_j^0 so long as we keep them bigger than c . Thus a_p, b_q is reachable from a_p^0, b_q^0 .

If $r = 0$ then the problem is slightly more involved. WLOG no parameter is zero. We now have one additional complication: no parameter can ever move through zero. Hence a_p, b_q is reachable from a_p^0, b_q^0 if and only if the number of $a_i < 0$ equals the number of $a_i^0 < 0$, and similarly for the b_i and b_i^0 . But in a suitable set of parameters, all $b_j > 0$! This is because the Hypergeometric function is undefined if one of the b_j is a non-positive integer and all a_i are smaller than the b_j . Hence the number of $b_j \leq 0$ is always zero.

We can thus associate to every suitable set of parameters a_p, b_q , where no $a_i = 0$, the following invariants:

- For every $r \in [0, 1)$ the number α_r of parameters $a_i \equiv r \pmod{1}$, and similarly the number β_r of parameters $b_i \equiv r \pmod{1}$.
- The number γ of integers a_i with $a_i < 0$.

The above reasoning shows that a_p, b_q is reachable from a_p^0, b_q^0 if and only if the invariants $\alpha_r, \beta_r, \gamma$ all agree. Thus in particular “being reachable from” is a symmetric relation on suitable parameters without zeros.

Applying the Operators

If all goes well then for a given set of parameters we find an origin in our database for which we have a nice formula. We now have to apply (potentially) many differential operators to it. If we do this blindly then the result will be very messy. This is because with Hypergeometric type functions, the derivative is usually expressed as a sum of two contiguous functions. Hence if we compute N derivatives, then the answer will involve $2N$ contiguous functions! This is clearly undesirable. In fact we know from the Hypergeometric differential equation that we need at most $\max(p, q + 1)$ contiguous functions to express all derivatives.

Hence instead of differentiating blindly, we will work with a $\mathbb{C}(z)$ -module basis: for an origin a_r^0, b_s^0 we either store (for particularly pretty answers) or compute a set of N functions (typically $N = \max(r, s + 1)$) with the property that the derivative of any of them is a $\mathbb{C}(z)$ -linear combination of them. In formulae, we store a vector B of N functions, a matrix M and a vector C (the latter two with entries in $\mathbb{C}(z)$), with the following properties:

- ${}_rF_s \left(\begin{smallmatrix} a_r^0 \\ b_s^0 \end{smallmatrix} \middle| z \right) = CB$
- $z \frac{d}{dz} B = MB$.

Then we can compute as many derivatives as we want and we will always end up with $\mathbb{C}(z)$ -linear combination of at most N special functions.

As hinted above, B , M and C can either all be stored (for particularly pretty answers) or computed from a single ${}_pF_q$ formula.

Loose Ends

This describes the bulk of the hypergeometric function algorithm. There a few further tricks, described in the `hyperexpand.py` source file. The extension to Meijer G-functions is also described there.

5.3.2 Meijer G-Functions of Finite Confluence

Slater's theorem essentially evaluates a G -function as a sum of residues. If all poles are simple, the resulting series can be recognised as hypergeometric series. Thus a G -function can be evaluated as a sum of Hypergeometric functions.

If the poles are not simple, the resulting series are not hypergeometric. This is known as the “confluent” or “logarithmic” case (the latter because the resulting series tend to contain logarithms). The answer depends in a complicated way on the multiplicities of various poles, and there is no accepted notation for representing it (as far as I know). However if there are only finitely many multiple poles, we can evaluate the G function as a sum of hypergeometric functions, plus finitely many extra terms. I could not find any good reference for this, which is why I work it out here.

Recall the general setup. We define

$$G(z) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where L is a contour starting and ending at $+\infty$, enclosing all of the poles of $\Gamma(b_j - s)$ for $j = 1, \dots, n$ once in the negative direction, and no other poles. Also the integral is assumed absolutely convergent.

In what follows, for any complex numbers a, b , we write $a \equiv b \pmod{1}$ if and only if there exists an integer k such that $a - b = k$. Thus there are double poles iff $a_i \equiv a_j \pmod{1}$ for some $i \neq j \leq n$.

We now assume that whenever $b_j \equiv a_i \pmod{1}$ for $i \leq m$, $j > n$ then $b_j < a_i$. This means that no quotient of the relevant gamma functions is a polynomial, and can always be achieved by “reduction of order”. Fix a complex number c such that $\{b_i | b_i \equiv c \pmod{1}, i \leq m\}$ is not empty. Enumerate this set as $b, b + k_1, \dots, b + k_u$, with k_i non-negative integers. Enumerate similarly $\{a_j | a_j \equiv c \pmod{1}, j > n\}$ as $b + l_1, \dots, b + l_v$. Then $l_i > k_j$ for all i, j . For finite confluence, we need to assume $v \geq u$ for all such c .

Let c_1, \dots, c_w be distinct $\pmod{1}$ and exhaust the congruence classes of the b_i . I claim

$$G(z) = - \sum_{j=1}^w (F_j(z) + R_j(z)),$$

where $F_j(z)$ is a hypergeometric function and $R_j(z)$ is a finite sum, both to be specified later. Indeed corresponding to every c_j there is a sequence of poles, at mostly finitely many of them multiple poles. This is where the j -th term comes from.

Hence fix again c , enumerate the relevant b_i as $b, b + k_1, \dots, b + k_u$. We will look at the a_j corresponding to $a + l_1, \dots, a + l_u$. The other a_i are not treated specially. The corresponding gamma functions have poles at (potentially) $s = b + r$ for $r = 0, 1, \dots$. For $r \geq l_u$, pole of the integrand is simple. We thus set

$$R(z) = \sum_{r=0}^{l_u-1} \text{res}_{s=r+b}.$$

We finally need to investigate the other poles. Set $r = l_u + t$, $t \geq 0$. A computation shows

$$\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)_{l_i - k_i}} = \frac{(-1)^{\delta_i}}{(l_u - l_i + 1)_{\delta_i}} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t},$$

where $\delta_i = l_i - k_i$.

Also

$$\Gamma(b_j - l_u - b - t) = \frac{\Gamma(b_j - l_u - b)}{(-1)^t (l_u + b + 1 - b_j)_t},$$

$$\Gamma(1 - a_j + l_u + b + t) = \Gamma(1 - a_j + l_u + b) (1 - a_j + l_u + b)_t$$

and

$$\text{res}_{s=b+l_u+t} \Gamma(b - s) = -\frac{(-1)^{l_u+t}}{(l_u + t)!} = -\frac{(-1)^{l_u}}{l_u!} \frac{(-1)^t}{(l_u + 1)_t}.$$

Hence

$$\begin{aligned} \text{res}_{s=b+l_u+t} &= -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^u \frac{(-1)^{\delta_i}}{(l_u - k_i + 1)_{\delta_i}} \frac{\prod_{j=1}^n \Gamma(1 - a_j + l_u + b) \prod_{j=1}^m \Gamma(b_j - l_u - b)^*}{\prod_{j=n+1}^p \Gamma(a_j - l_u - b)^* \prod_{j=m+1}^q \Gamma(1 - b_j + l_u + b)} \\ &\times z^t \frac{(-1)^t}{(l_u + 1)_t} \prod_{i=1}^u \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t} \frac{\prod_{j=1}^n (1 - a_j + l_u + b)_t \prod_{j=n+1}^p (-1)^t (l_u + b + 1 - a_j)_t^*}{\prod_{j=1}^m (-1)^t (l_u + b + 1 - b_j)_t^* \prod_{j=m+1}^q (1 - b_j + l_u + b)_t}, \end{aligned}$$

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.)

5.3.3 Extending The Hypergeometric Tables

Adding new formulae to the tables is straightforward. At the top of the file `diofant/simplify/hyperexpand.py`, there is a function called `add_formulae()` (page 798). Nested in it are defined two helpers, `add(ap, bq, res)` and `addb(ap, bq, B, C, M)`, as well as dummies `a`, `b`, `c`, and `z`.

The first step in adding a new formula is by using `add(ap, bq, res)`. This declares `hyper(ap, bq, z) == res`. Here `ap` and `bq` may use the dummies `a`, `b`, and `c` as free symbols. For example the well-known formula $\sum_0^\infty \frac{(-a)_n z^n}{n!} = (1-z)^a$ is declared by the following line: `add((-a,), (), (1-z)**a)`.

From the information provided, the matrices B , C and M will be computed, and the formula is now available when expanding hypergeometric functions. Next the test file `diofant/simplify/tests/test_hyperexpand.py` should be run, in particular the test `test_formulae`. This will test the newly added formula numerically. If it fails, there is (presumably) a typo in what was entered.

Since all newly-added formulae are probably relatively complicated, chances are that the automatically computed basis is rather suboptimal (there is no good way of testing this, other than observing very messy output). In this case the matrices B , C and M should be computed by hand. Then the helper `addb` can be used to declare a hypergeometric formula with hand-computed basis.

An example

Because this explanation so far might be very theoretical and difficult to understand, we walk through an explicit example now. We take the Fresnel function $C(z)$ which obeys the following hypergeometric representation:

$$C(z) = z \cdot {}_1F_2 \left(\frac{1}{4} \middle| - \frac{\pi^2 z^4}{16} \right).$$

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 ${}_1F_2(\dots|f(z))$ with f a function of z , in our case

$$f(z) = -\frac{\pi^2 z^4}{16}.$$

What we need is a formula where the hypergeometric function has only z as argument ${}_1F_2(\dots|z)$. We introduce the new complex symbol w and search for a function $g(w)$ such that

$$f(g(w)) = w$$

holds. Then we can replace every z in $C(z)$ by $g(w)$. In the case of our example the function g could look like

$$g(w) = \frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}.$$

We get these functions mainly by guessing and testing the result. Hence we proceed by computing $f(g(w))$ (and simplifying naively)

$$\begin{aligned} f(g(w)) &= -\frac{\pi^2 g(w)^4}{16} \\ &= -\frac{\pi^2 g \left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}} \right)^4}{16} \\ &= -\frac{\pi^2 \frac{2^4}{\sqrt{\pi}^4} \exp\left(\frac{i\pi}{4}\right)^4 w^{\frac{1}{4}^4}}{16} \\ &= -\exp(i\pi) w \\ &= w \end{aligned}$$

and indeed get back w . (In case of branched functions we have to be aware of branch cuts. In that case we take w to be a positive real number and check the formula. If what we have found works for positive w , then just replace `exp()` (page 295) inside any branched function by `exp_polar()` (page 295) and what we get is right for *all* w .) Hence we can write the formula as

$$C(g(w)) = g(w) \cdot {}_1F_2 \left(\frac{1}{2}, \frac{5}{4} \middle| w \right).$$

and trivially

$${}_1F_2 \left(\frac{1}{2}, \frac{5}{4} \middle| w \right) = \frac{C(g(w))}{g(w)} = \frac{C \left(\frac{2}{\sqrt{\pi}} \exp \left(\frac{i\pi}{4} \right) w^{\frac{1}{4}} \right)}{\frac{2}{\sqrt{\pi}} \exp \left(\frac{i\pi}{4} \right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third parameter, `res`, in `add`. Finally, the whole function call to add this rule to the table looks like:

```
add([Rational(1, 4)],
    [Rational(1, 2), Rational(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 $\{B_0, B_1, \dots\}$ and is of shape $n \times 1$. The best way to get B_i is to take the first $n = \max(p, q + 1)$ derivatives of the expression for ${}_pF_q$ and take out useful pieces. In our case we find that $n = \max(1, 2 + 1) = 3$. For computing the derivatives, we have to use the operator $z \frac{d}{dz}$. The first basis element B_0 is set to the expression for ${}_1F_2$ from above:

$$B_0 = \frac{\sqrt{\pi} \exp \left(-\frac{i\pi}{4} \right) C \left(\frac{2}{\sqrt{\pi}} \exp \left(\frac{i\pi}{4} \right) z^{\frac{1}{4}} \right)}{2z^{\frac{1}{4}}}$$

Next we compute $z \frac{d}{dz} B_0$. For this we can directly use Diofant!

```
>>> 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) - E**(-I*pi/4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*root(z, 4)/
->sqrt(pi))/(8*root(z, 4)**5))
>>> expand(_)
cosh(2*sqrt(z))/4 - E**(-I*pi/4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*root(z, 4)/sqrt(pi))/
->(8*root(z, 4))
```

Formatting this result nicely we obtain

$$B'_1 = -\frac{1}{4} \frac{\sqrt{\pi} \exp \left(-\frac{i\pi}{4} \right) C \left(\frac{2}{\sqrt{\pi}} \exp \left(\frac{i\pi}{4} \right) z^{\frac{1}{4}} \right)}{2z^{\frac{1}{4}}} + \frac{1}{4} \cosh(2\sqrt{z})$$

Computing the second derivative we find

```
>>> Blprime = 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(Blprime, z)
z*(-cosh(2*sqrt(z))/(16*z) + sinh(2*sqrt(z))/(4*sqrt(z)) + E**(-I*pi/
↪4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*root(z, 4)/sqrt(pi))/(32*root(z, 4)**5))
>>> expand(_)
sqrt(z)*sinh(2*sqrt(z))/4 - cosh(2*sqrt(z))/16 + E**(-I*pi/
↪4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*root(z, 4)/sqrt(pi))/(32*root(z, 4))
```

which can be printed as

$$B'_2 = \frac{1}{16} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16} \cosh(2\sqrt{z}) + \frac{1}{4} \sinh(2\sqrt{z})\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose B_1 and B_2 as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} \\ \cosh(2\sqrt{z}) \\ \sinh(2\sqrt{z})\sqrt{z} \end{pmatrix}$$

(This is in contrast to the basis $B = (B_0, B'_1, B'_2)$ that would have been computed automatically if we used just `add(ap, bq, res)`.)

Because it must hold that ${}_pF_q(\cdots|z) = CB$ the entries of C are obviously

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Finally we have to compute the entries of the 3×3 matrix M such that $z \frac{d}{dz} B = MB$ holds. This is easy. We already computed the first part $z \frac{d}{dz} B_0$ above. This gives us the first row of M . For the second row we have:

```
>>> B1 = cosh(2*sqrt(z))
>>> z * diff(B1, z)
sqrt(z)*sinh(2*sqrt(z))
```

and for the third one

```
>>> 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`.

5.3.4 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 Diofant (of course many more are derived from them). These formulae are mostly taken from [Luk69] and [APPM90]. They are all tested numerically.

$$\begin{aligned}
 {}_0F_0(|z) &= e^z \\
 {}_1F_0(a|z) &= (-z+1)^{-a} \\
 {}_2F_1\left(a, a-\frac{1}{2} \middle| z\right) &= 2^{2a-1} (\sqrt{-z+1}+1)^{-2a+1} \\
 {}_2F_1\left(1, 1 \middle| z\right) &= -\frac{1}{z} \log(-z+1) \\
 {}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{1}{\sqrt{z}} \operatorname{atanh}(\sqrt{z}) \\
 {}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{1}{\sqrt{z}} \operatorname{asin}(\sqrt{z}) \\
 {}_2F_1\left(a, a+\frac{1}{2} \middle| z\right) &= \frac{1}{2} (\sqrt{z}+1)^{-2a} + \frac{1}{2} (-\sqrt{z}+1)^{-2a} \\
 {}_2F_1\left(a, -a \middle| z\right) &= \cos(2a \operatorname{asin}(\sqrt{z})) \\
 {}_2F_1\left(1, 1 \middle| z\right) &= \frac{\operatorname{asin}(\sqrt{z})}{\sqrt{z}\sqrt{-z+1}} \\
 {}_2F_1\left(\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{2K(z)}{\pi} \\
 {}_2F_1\left(-\frac{1}{2}, \frac{1}{2} \middle| z\right) &= \frac{2E(z)}{\pi} \\
 {}_3F_2\left(-\frac{1}{2}, 1, 1 \middle| z\right) &= -\frac{2\sqrt{z}}{3} \operatorname{atanh}(\sqrt{z}) + \frac{2}{3} - \frac{1}{3z} \log(-z+1) \\
 {}_3F_2\left(-\frac{1}{2}, 1, 1 \middle| 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(\frac{1}{b} \middle| z\right) &= e^z z^{-b+1} (b-1) \gamma(b-1, z) \\
 {}_1F_1\left(\frac{a}{2a} \middle| z\right) &= 4^{a-\frac{1}{2}} e^{\frac{z}{2}} z^{-a+\frac{1}{2}} I_{a-\frac{1}{2}}\left(\frac{z}{2}\right) \Gamma\left(a+\frac{1}{2}\right) \\
 {}_1F_1\left(\frac{a}{a+1} \middle| z\right) &= a (z \exp_{\text{polar}}(i\pi))^{-a} \gamma(a, z \exp_{\text{polar}}(i\pi)) \\
 {}_1F_1\left(-\frac{1}{2} \middle| z\right) &= e^z + i\sqrt{\pi}\sqrt{z} \operatorname{erf}(i\sqrt{z}) \\
 {}_1F_2\left(\frac{1}{4}, \frac{5}{4} \middle| z\right) &= \frac{e^{-\frac{i\pi}{4}} \sqrt{\pi}}{2\sqrt[4]{z}} \left(i \sinh(2\sqrt{z}) S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) + \cosh(2\sqrt{z}) C\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) \right) \\
 {}_2F_2\left(\frac{1}{2}, \frac{1}{2}, a \middle| z\right) &= -\frac{i\sqrt{\pi}a\sqrt{\frac{1}{z}}}{2a-1} \operatorname{erf}(i\sqrt{z}) - \frac{a (z \exp_{\text{polar}}(i\pi))^{-a}}{2a-1} \gamma(a, z \exp_{\text{polar}}(i\pi))
 \end{aligned}$$

$$\begin{aligned}
{}_2F_2 \left(\begin{matrix} 1, 1 \\ 2, 2 \end{matrix} \middle| z \right) &= \frac{1}{z} (-\log(z) + \text{Ei}(z)) - \frac{\gamma}{z} \\
{}_0F_1 \left(\begin{matrix} \\ \frac{1}{2} \end{matrix} \middle| z \right) &= \cosh(2\sqrt{z}) \\
{}_0F_1 \left(\begin{matrix} \\ b \end{matrix} \middle| z \right) &= z^{-\frac{b}{2} + \frac{1}{2}} I_{b-1}(2\sqrt{z}) \Gamma(b) \\
{}_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}} (I_{2a-1}(4\sqrt[4]{z}) + J_{2a-1}(4\sqrt[4]{z})) \Gamma(2a) \\
{}_0F_3 \left(\begin{matrix} \\ a, a + \frac{1}{2}, 2a \end{matrix} \middle| z \right) &= \left(2\sqrt{z} \exp_{\text{polar}} \left(\frac{i\pi}{2} \right) \right)^{-2a+1} I_{2a-1} \left(2\sqrt{2}\sqrt[4]{z} \exp_{\text{polar}} \left(\frac{i\pi}{4} \right) \right) J_{2a-1} \left(2\sqrt{2}\sqrt[4]{z} \exp_{\text{polar}} \left(\frac{i\pi}{4} \right) \right) \Gamma^2(2a) \\
{}_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}}(\sqrt{z}) I_{a-\frac{1}{2}}(\sqrt{z}) \Gamma\left(a - \frac{1}{2}\right) \Gamma\left(a + \frac{1}{2}\right) - 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} I_{a-\frac{1}{2}}^2(\sqrt{z}) \Gamma^2\left(a + \frac{1}{2}\right) \\
{}_1F_2 \left(\begin{matrix} \frac{1}{2} \\ b, -b + 2 \end{matrix} \middle| z \right) &= \frac{\pi I_{-b+1}(\sqrt{z}) I_{b-1}(\sqrt{z})}{\sin(\pi b)} (-b + 1) \\
{}_1F_2 \left(\begin{matrix} \frac{1}{2} \\ \frac{3}{2}, \frac{3}{2} \end{matrix} \middle| z \right) &= \frac{1}{2\sqrt{z}} \text{Shi}(2\sqrt{z}) \\
{}_1F_2 \left(\begin{matrix} \frac{3}{2} \\ \frac{3}{2}, \frac{7}{4} \end{matrix} \middle| z \right) &= \frac{3\sqrt{\pi}}{4z^{\frac{3}{4}}} e^{-\frac{3i}{4}\pi} S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) \\
{}_1F_2 \left(\begin{matrix} \frac{1}{2} \\ \frac{1}{2}, \frac{5}{4} \end{matrix} \middle| z \right) &= \frac{e^{-\frac{i\pi}{4}} \sqrt{\pi}}{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}(\sqrt{z}) I_{b-1}(\sqrt{z}) \Gamma(b) \Gamma(2a - b + 1) \\
{}_2F_3 \left(\begin{matrix} 1, 1 \\ 2, 2, \frac{3}{2} \end{matrix} \middle| z \right) &= \frac{1}{z} (-\log(2\sqrt{z}) + \text{Chi}(2\sqrt{z})) - \frac{\gamma}{z} \\
{}_3F_3 \left(\begin{matrix} 1, 1, a \\ 2, 2, a + 1 \end{matrix} \middle| z \right) &= -\frac{e^z a}{z(a^2 - 2a + 1)} + \frac{a(-z)^{-a}}{(a-1)^2} (\Gamma(a) - \Gamma(a, -z)) + \frac{a}{z(a^2 - 2a + 1)} (-a + 1) (\log(-z) + \text{E}_1(-z) + \gamma) +
\end{aligned}$$

`diofant.simplify.hyperexpand.add_formulae(formulae)`

Create our knowledge base.

5.4 Computing Integrals using Meijer G-Functions

This text aims to describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

5.4.1 Overview

The algorithm to compute $\int f(x)dx$ or $\int_0^\infty f(x)dx$ generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function `hyperexpand` (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g. $\int_{-\infty}^\infty$) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.

5.4.2 Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being $0, \pm 1, \infty$). This is not very important when e.g. expanding a single hypergeometric function into named special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set $S = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : S \rightarrow \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives S both a topology and a complex structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps $\text{Exp} : \mathbb{C} \rightarrow S, (x + iy) \mapsto (\exp(x), y)$ and $\text{Log} : S \rightarrow \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point “zero” (0) to S and denote the resulting object S_0 . Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by $\{\text{Exp}(z) : \Re(z) < k\}$, which at least defines a topology. Elements of S_0 shall be called polar numbers. We further define functions $\text{Arg} : S \rightarrow \mathbb{R}, (r, \theta) \mapsto \theta$ and $|\cdot| : S_0 \rightarrow \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from \mathbb{C} to S . We define $\text{Exp}(a)\text{Exp}(b) = \text{Exp}(a + b)$ for $a, b \in \mathbb{C}$, also for $a \in S$ and $b \in \mathbb{C}$ we define $a^b = \text{Exp}(b \text{Log}(a))$. It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g. $(ab)^c = a^c b^c$) which hold in \mathbb{C} only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions $F(z) = \sqrt{1+z}$ and $G(a, b) = \sqrt{a+b}$, where a, b, z are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if $S(z)$ denotes the standard branch of the square root function on \mathbb{C} , we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \text{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} . \\ -S(p(z)) & : \text{else} \end{cases}$$

(We are omitting $|z| = 1$ here, this does not matter for integration.) Finally we define $G(a, b) = \sqrt{a}F(b/a)$.

5.4.3 Representing Branched Functions on the Argand Plane

Suppose $f : \mathcal{S} \rightarrow \mathbb{C}$ is a holomorphic function. We wish to define a function F on (part of) the complex numbers \mathbb{C} that represents f as closely as possible. This process is known as “introducing branch cuts”. In our situation, there is actually a canonical way of doing this (which is adhered to in all of Diofant), as follows: Introduce the “cut complex plane” $C = \mathbb{C} \setminus \mathbb{R}_{\leq 0}$. Define a function $l : C \rightarrow \mathcal{S}$ 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 is called “lifting to the principal branch” throughout the Diofant documentation.

5.4.4 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 occurring in $f_i(x)$ and (b) the number of different functions occurring in $f_i(x)$. Thus given e.g. $f(x) = \sin x e^x \sin 2x$ we should try $f_1(x) = \sin x \sin 2x$, $f_2(x) = e^x$ first. All of this is done by the function `_mul_as_two_parts(f)`.

Finally, we can try a recursive Mellin transform technique. Since the Meijer G-function is defined essentially as a certain inverse mellin transform, if we want to write a function $f(x)$ as a G-function, we can compute its mellin transform $F(s)$. If $F(s)$ is in the right form, the G-function expression can be read off. This technique generalises many standard rewritings, e.g. $e^{ax}e^{bx} = e^{(a+b)x}$.

One twist is that some functions don’t have mellin transforms, even though they can be written as G-functions. This is true for example for $f(x) = e^x \sin x$ (the function grows too rapidly to have a mellin transform). However if the function is recognised to be analytic, then we can try to compute the mellin-transform of $f(ax)$ for a parameter a , and deduce the G-function expression by analytic continuation. (Checking for analyticity is easy. Since we can only deal with a certain subset of functions anyway, we only have to filter out those which are not analytic.)

The function `_rewrite_single` does the table lookup and recursive mellin transform. The functions `_rewrite1` and `_rewrite2` respectively use above-mentioned helpers and `_rewrite_single` to rewrite their argument as respectively one or two G-functions.

5.4.5 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](#)).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing e^z satisfies $G(\text{Exp}(2\pi i)z) = G(z)$ for all $z \in \mathcal{S}$. The function `meijerg.get_period()` can be used to discover this, and the function `principal_branch(z, period)` in `functions/elementary/complexes.py` can be used to exploit the information. This is done transparently by the integration code.

5.4.6 The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually stated in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway.):

1.

$$\int_0^\infty G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \eta x \right) dx = \frac{\prod_{j=1}^m \Gamma(b_j + 1) \prod_{j=1}^n \Gamma(-a_j)}{\eta \prod_{j=m+1}^q \Gamma(-b_j) \prod_{j=n+1}^p \Gamma(a_j + 1)}$$

2.

$$\int_0^\infty G_{u,v}^{s,t} \left(\begin{matrix} c_1, \dots, c_u \\ d_1, \dots, d_v \end{matrix} \middle| \sigma x \right) G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \omega x \right) dx = G_{v+p, u+q}^{m+t, n+s} \left(\begin{matrix} a_1, \dots, a_n, -d_1, \dots, -d_v, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, -c_1, \dots, -c_u, b_{m+1}, \dots, b_q \end{matrix} \middle| \frac{\omega}{\sigma} \right)$$

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in Diofant. They are an amalgamation of conditions found in [APPM90] and [Luk69]; 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. [Luk69], section 5.3). We introduce the following notation:

$$\begin{aligned} \xi &= m + n - p \\ \delta &= m + n - \frac{p + q}{2} \\ C_3 &: -\Re(b_j) < 1 \text{ for } j = 1, \dots, m \\ &0 < -\Re(a_j) \text{ for } j = 1, \dots, n \\ C_3^* &: -\Re(b_j) < 1 \text{ for } j = 1, \dots, q \\ &0 < -\Re(a_j) \text{ for } j = 1, \dots, p \end{aligned}$$

$$C_4 : -\Re(\delta) + \frac{q+1-p}{2} > q-p$$

The convergence conditions will be detailed in several “cases”, numbered one to five. For later use it will be helpful to separate conditions “at infinity” from conditions “at zero”. By conditions “at infinity” we mean conditions that only depend on the behaviour of the integrand for large, positive values of x , whereas by conditions “at zero” we mean conditions that only depend on the behaviour of the integrand on $(0, \epsilon)$ for any $\epsilon > 0$. Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions n “at zero” and “at infinity” both have to be fulfilled, for some n .

These are the conditions “at infinity”:

1.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi \wedge (A \vee B \vee C),$$

where

$$A = 1 \leq n \wedge p < q \wedge 1 \leq m$$

$$B = 1 \leq p \wedge 1 \leq m \wedge q = p+1 \wedge \neg(n=0 \wedge m=p+1)$$

$$C = 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \dots \left\lceil \frac{\delta}{2} \right\rceil.$$

2.

$$n = 0 \wedge p+1 \leq m \wedge |\arg(\eta)| < \delta\pi$$

3.

$$(p < q \wedge 1 \leq m \wedge \delta > 0 \wedge |\arg(\eta)| = \delta\pi) \vee (p \leq q-2 \wedge \delta = 0 \wedge \arg(\eta) = 0)$$

4.

$$p = q \wedge \delta = 0 \wedge \arg(\eta) = 0 \wedge \eta \neq 0 \wedge \Re \left(\sum_{j=1}^p b_j - a_j \right) < 0$$

5.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi$$

And these are the conditions “at zero”:

1.

$$\eta \neq 0 \wedge C_3$$

2.

$$C_3$$

3.

$$C_3 \wedge C_4$$

4.

$$C_3$$

5.

$$C_3$$

Conditions of Convergence for Integral (2)

We introduce the following notation:

$$\begin{aligned}
 b^* &= s + t - \frac{u + v}{2} \\
 c^* &= m + n - \frac{p + q}{2} \\
 \rho &= \sum_{j=1}^v d_j - \sum_{j=1}^u c_j + \frac{u - v}{2} + 1 \\
 \mu &= \sum_{j=1}^q b_j - \sum_{j=1}^p a_j + \frac{p - q}{2} + 1 \\
 \phi &= q - p - \frac{u - v}{2} + 1 \\
 \eta &= 1 - (v - u) - \mu - \rho \\
 \psi &= \frac{\pi(q - m - n) + |\arg(\omega)|}{q - p} \\
 \theta &= \frac{\pi(v - s - t) + |\arg(\sigma)|}{v - u} \\
 \lambda_c &= (q - p)|\omega|^{1/(q-p)} \cos \psi + (v - u)|\sigma|^{1/(v-u)} \cos \theta \\
 \lambda_{s0}(c_1, c_2) &= c_1(q - p)|\omega|^{1/(q-p)} \sin \psi + c_2(v - u)|\sigma|^{1/(v-u)} \sin \theta \\
 \lambda_s &= \begin{cases} \lambda_{s0}(-1, -1) \lambda_{s0}(1, 1) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), -1) \lambda_{s0}(\text{sign}(\arg(\omega)), 1) & \text{for } \arg(\omega) \neq 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(-1, \text{sign}(\arg(\sigma))) \lambda_{s0}(1, \text{sign}(\arg(\sigma))) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) \neq 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), \text{sign}(\arg(\sigma))) & \text{otherwise} \end{cases} \\
 z_0 &= \frac{\omega}{\sigma} e^{-i\pi(b^* + c^*)} \\
 z_1 &= \frac{\sigma}{\omega} e^{-i\pi(b^* + c^*)}
 \end{aligned}$$

The following conditions will be helpful:

$$\begin{aligned}
 C_1 : & (a_i - b_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, n, j = 1, \dots, m) \\
 & \wedge (c_i - d_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, t, j = 1, \dots, s)
 \end{aligned}$$

$$C_2 : \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \dots, m, j = 1, \dots, s$$

$$C_3 : \Re(a_i + c_j) < 1 \text{ for } i = 1, \dots, n, j = 1, \dots, t$$

$$C_4 : (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, t$$

$$C_5 : (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, s$$

$$C_6 : (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, n$$

$$C_7 : (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, m$$

$$C_8 : 0 < |\phi| + 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))$$

$$C_9 : 0 < |\phi| - 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))$$

$$C_{10} : |\arg(\sigma)| < \pi b^*$$

$$C_{11} : |\arg(\sigma)| = \pi b^*$$

$$C_{12} : |\arg(\omega)| < c^* \pi$$

$$C_{13} : |\arg(\omega)| = c^* \pi$$

$$C_{14}^1 : (z_0 \neq 1 \wedge |\arg(1 - z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu + \rho - u + v) < 1)$$

$$C_{14}^2 : (z_1 \neq 1 \wedge |\arg(1 - z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu + \rho - p + q) < 1)$$

$$C_{14} : \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2)$$

$$C_{15} : \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0)$$

$$C_{16} : \int_0^\infty G_{u,v}^{s,t}(\sigma x) dx \text{ converges at infinity}$$

$$C_{17} : \int_0^\infty G_{p,q}^{m,n}(\omega x) dx \text{ converges at infinity}$$

Note that C_{16} and C_{17} are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1.

$$mnst \neq 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

2.

$$u = v \wedge b^* = 0 \wedge 0 < c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{12}$$

3.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re \mu < 1 \wedge \Re \rho < 1 \wedge \sigma \neq \omega \wedge C_1 \wedge C_2 \wedge C_3$$

4.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

6.

$$q < p \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{10} \wedge C_{13}$$

7.

$$p < q \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{10} \wedge C_{13}$$

8.

$$v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12}$$

9.

$$u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12}$$

10.

$$q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13}$$

11.

$$p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13}$$

12.

$$p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11}$$

13.

$$p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11}$$

14.

$$p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13}$$

15.

$$q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13}$$

16.

$$q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

17.

$$p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

18.

$$t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10}$$

19.

$$s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10}$$

20.

$$n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12}$$

21.

$$m = 0 \wedge 0 < n \wedge 0 < c^* \wedge 0 < \phi \wedge C_1 \wedge C_3 \wedge C_{12}$$

22.

$$st = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

23.

$$mn = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

24.

$$p < m + n \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

25.

$$q < m + n \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

26.

$$p = q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

27.

$$p = q + 1 \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

28.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

30.

$$n = 0 \wedge \phi = 0 \wedge 0 < s + t \wedge 0 < m \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

31.

$$m = 0 \wedge \phi = 0 \wedge v < s + t \wedge 0 < n \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

32.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

36.

$$C_{17} \wedge t = 0 \wedge u < s \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

37.

$$C_{17} \wedge s = 0 \wedge v < t \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

38.

$$C_{16} \wedge n = 0 \wedge p < m \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

39.

$$C_{16} \wedge m = 0 \wedge q < n \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

5.4.7 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 [Luk69], section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument bz , and convergence conditions are also given. However, we need a formula for argument bz^a for rational a .

We are asked to compute

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a) dz,$$

for positive real t . Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

How to compute the integral

We shall work formally for now. Denote by $\Delta(s)$ the product of gamma functions appearing in the definition of G , so that

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s) z^s ds.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} ds dz.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{dz}{2\pi i} ds.$$

The inner integral is easily seen to be $\frac{1}{\Gamma(-as)} \frac{1}{t^{1+as}}$. (Using Cauchy's theorem and Jordan's lemma deform the contour to run from $-\infty$ to $-\infty$, encircling 0 once in the negative sense. For as real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

$$f(t) = \frac{1}{t} \frac{1}{2\pi i} \int_L \Delta(s) \frac{1}{\Gamma(-as)} \left(\frac{b}{t^a}\right)^s ds,$$

which is a so-called Fox H function (of argument $\frac{b}{t^a}$). For rational a , this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for $\Re(as) < -1$. Thus the contour L has to run left of the line $\Re(as) = -1$. Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient b . Hence so is $f(t)$. Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that $a > 0$, then the latter condition is fulfilled if G converges along contours (2) or (3) of [Luk69], section 5.2, i.e. either $\delta \geq \frac{a}{2}$ or $p \geq 1, p \geq q$.

When the integral exists

Using [Luk69], section 5.10, for any given meijer G-function we can find a dominant term of the form $z^a e^{bz^c}$ (although this expression might not be the best possible, because of cancellation).

We must thus investigate

$$\lim_{T \rightarrow \infty} \int_{c-iT}^{c+iT} e^{zt} z^a e^{bz^c} dz.$$

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write $z = c + i\tau$. Then $\arg(z) \rightarrow \pm \frac{\pi}{2}$, and so $e^{zt} \sim e^{it\tau}$ (where \sim shall always mean “asymptotically equivalent up to a positive real multiplicative constant”). Also $z^{x+iy} \sim |\tau|^x e^{iy \log |\tau|} e^{\pm xi \frac{\pi}{2}}$.

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.

5.4.8 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 Diofant. An entry “generated” means that the corresponding G-function has a variable number of parameters. This table is intended to shrink in future, when the algorithm’s capabilities of deriving new formulae improve. Of course it has to grow whenever a new class of special functions is to be dealt with. Elementary functions:

$$\begin{aligned} a &= aG_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) + aG_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \\ (b + pz^q)^{-a} &= \frac{b^{-a}}{\Gamma(a)} G_{1,1}^{1,1} \left(\begin{matrix} -a+1 \\ 0 \end{matrix} \middle| \frac{pz^q}{b} \right) \\ \frac{-b^a + (pz^q)^a}{-b + pz^q} &= \frac{1}{\pi} b^{a-1} G_{2,2}^{2,2} \left(\begin{matrix} 0, a \\ 0, a \end{matrix} \middle| \frac{pz^q}{b} \right) \sin(\pi a) \\ \frac{-b^a + z^a}{-b + z} &= \frac{1}{\pi} b^{a-1} G_{2,2}^{2,2} \left(\begin{matrix} 0, a \\ 0, a \end{matrix} \middle| \frac{z}{b} \right) \sin(\pi a) \\ (a + \sqrt{a^2 + pz^q})^b &= -\frac{a^b b}{2\sqrt{\pi}} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ 0 \end{matrix} \middle| \frac{pz^q}{a^2} \right) \\ (-a + \sqrt{a^2 + pz^q})^b &= \frac{a^b b}{2\sqrt{\pi}} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ b \end{matrix} \middle| \frac{pz^q}{a^2} \right) \\ \frac{(a + \sqrt{a^2 + pz^q})^b}{\sqrt{a^2 + pz^q}} &= \frac{1}{\sqrt{\pi}} a^{b-1} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ 0 \end{matrix} \middle| \frac{pz^q}{a^2} \right) \\ \frac{(-a + \sqrt{a^2 + pz^q})^b}{\sqrt{a^2 + pz^q}} &= \frac{1}{\sqrt{\pi}} a^{b-1} G_{2,2}^{1,2} \left(\begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ b \end{matrix} \middle| \frac{pz^q}{a^2} \right) \end{aligned}$$

$$\begin{aligned} \left(\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q}\right)^b &= -\frac{a^{\frac{b}{2}}b}{2\sqrt{\pi}}G_{2,2}^{2,1}\left(\begin{matrix} \frac{b}{2}+1 & -\frac{b}{2}+1 \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{pz^q}{a}\right) \\ \left(-\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q}\right)^b &= \frac{a^{\frac{b}{2}}b}{2\sqrt{\pi}}G_{2,2}^{2,1}\left(\begin{matrix} -\frac{b}{2}+1 & \frac{b}{2}+1 \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{pz^q}{a}\right) \\ \frac{\left(\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q}\right)^b}{\sqrt{a + pz^q}} &= \frac{1}{\sqrt{\pi}}a^{\frac{b}{2}-\frac{1}{2}}G_{2,2}^{2,1}\left(\begin{matrix} \frac{b}{2}+\frac{1}{2} & -\frac{b}{2}+\frac{1}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{pz^q}{a}\right) \\ \frac{\left(-\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q}\right)^b}{\sqrt{a + pz^q}} &= \frac{1}{\sqrt{\pi}}a^{\frac{b}{2}-\frac{1}{2}}G_{2,2}^{2,1}\left(\begin{matrix} -\frac{b}{2}+\frac{1}{2} & \frac{b}{2}+\frac{1}{2} \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{pz^q}{a}\right) \\ e^{pz^q} &= G_{0,1}^{1,0}\left(0 \middle| pz^q \exp_{\text{polar}}(-i\pi)\right) \end{aligned}$$

Functions involving $|b - pz^q|$:

$$|b - pz^q|^{-a} = 2G_{2,2}^{1,1}\left(\begin{matrix} -a+1 & -\frac{a}{2}+\frac{1}{2} \\ 0 & -\frac{a}{2}+\frac{1}{2} \end{matrix} \middle| \frac{pz^q}{b}\right) \sin\left(\frac{\pi a}{2}\right) |b|^{-a} \Gamma(-a+1), \text{ if } \Re a < 1$$

Functions involving $\text{Chi}(pz^q)$:

$$\text{Chi}(pz^q) = -\frac{\pi^{\frac{3}{2}}}{2}G_{2,4}^{2,0}\left(0, 0 \middle| \frac{1}{2}, \frac{1}{2} \middle| \frac{p^2}{4}z^{2q}\right)$$

Functions involving $\text{Ci}(pz^q)$:

$$\text{Ci}(pz^q) = -\frac{\sqrt{\pi}}{2}G_{1,3}^{2,0}\left(0, 0 \middle| \frac{1}{2} \middle| \frac{p^2}{4}z^{2q}\right)$$

Functions involving $\text{Ei}(pz^q)$:

$$\text{Ei}(pz^q) = -i\pi G_{1,1}^{1,0}\left(0 \middle| 1 \middle| z\right) - G_{1,2}^{2,0}\left(0, 0 \middle| 1 \middle| pz^q \exp_{\text{polar}}(i\pi)\right) - i\pi G_{1,1}^{0,1}\left(1 \middle| 0 \middle| z\right)$$

Functions involving $\theta(-b + pz^q)$:

$$\begin{aligned} (-b + pz^q)^{a-1} \theta(-b + pz^q) &= b^{a-1}G_{1,1}^{0,1}\left(a \middle| 0 \middle| \frac{pz^q}{b}\right) \Gamma(a), \text{ if } b > 0 \\ (b - pz^q)^{a-1} \theta(b - pz^q) &= b^{a-1}G_{1,1}^{1,0}\left(0 \middle| a \middle| \frac{pz^q}{b}\right) \Gamma(a), \text{ if } b > 0 \\ (-b + pz^q)^{a-1} \theta\left(z - \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) &= b^{a-1}G_{1,1}^{0,1}\left(a \middle| 0 \middle| \frac{pz^q}{b}\right) \Gamma(a), \text{ if } b > 0 \\ (b - pz^q)^{a-1} \theta\left(-z + \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) &= b^{a-1}G_{1,1}^{1,0}\left(0 \middle| a \middle| \frac{pz^q}{b}\right) \Gamma(a), \text{ if } b > 0 \end{aligned}$$

Functions involving $\text{Shi}(pz^q)$:

$$\text{Shi}(pz^q) = \frac{\sqrt{\pi}p}{4}z^q G_{1,3}^{1,1}\left(\frac{1}{2} \middle| -\frac{1}{2}, -\frac{1}{2} \middle| \frac{p^2}{4}z^{2q} \exp_{\text{polar}}(i\pi)\right)$$

Functions involving $\text{Si}(pz^q)$:

$$\text{Si}(pz^q) = \frac{\sqrt{\pi}}{2}G_{1,3}^{1,1}\left(\frac{1}{2} \middle| 0, 0 \middle| \frac{p^2}{4}z^{2q}\right)$$

Functions involving $I_a(pz^q)$:

$$I_a(pz^q) = \pi G_{1,3}^{1,0} \left(\begin{matrix} \frac{a}{2} & -\frac{\frac{a}{2} + \frac{1}{2}}{2} \\ \frac{a}{2} & -\frac{\frac{a}{2} + \frac{1}{2}}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $J_a(pz^q)$:

$$J_a(pz^q) = G_{0,2}^{1,0} \left(\begin{matrix} \frac{a}{2} & -\frac{a}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $K_a(pz^q)$:

$$K_a(pz^q) = \frac{1}{2} G_{0,2}^{2,0} \left(\begin{matrix} \frac{a}{2}, -\frac{a}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $Y_a(pz^q)$:

$$Y_a(pz^q) = G_{1,3}^{2,0} \left(\begin{matrix} \frac{a}{2}, -\frac{a}{2} & -\frac{\frac{a}{2} - \frac{1}{2}}{2} \\ \frac{a}{2}, -\frac{a}{2} & -\frac{\frac{a}{2} - \frac{1}{2}}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\cos(pz^q)$:

$$\cos(pz^q) = \sqrt{\pi} G_{0,2}^{1,0} \left(\begin{matrix} 0 & \frac{1}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\cosh(pz^q)$:

$$\cosh(pz^q) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(\begin{matrix} 0 & \frac{1}{2}, \frac{1}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $E(pz^q)$:

$$E(pz^q) = -\frac{1}{4} G_{2,2}^{1,2} \left(\begin{matrix} \frac{1}{2}, \frac{3}{2} \\ 0 \end{matrix} \middle| -pz^q \right)$$

Functions involving $K(pz^q)$:

$$K(pz^q) = \frac{1}{2} G_{2,2}^{1,2} \left(\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ 0 \end{matrix} \middle| -pz^q \right)$$

Functions involving $\operatorname{erf}(pz^q)$:

$$\operatorname{erf}(pz^q) = \frac{1}{\sqrt{\pi}} G_{1,2}^{1,1} \left(\begin{matrix} 1 \\ \frac{1}{2} \end{matrix} \middle| p^2 z^{2q} \right)$$

Functions involving $\operatorname{erfc}(pz^q)$:

$$\operatorname{erfc}(pz^q) = \frac{1}{\sqrt{\pi}} G_{1,2}^{2,0} \left(\begin{matrix} 1 \\ 0, \frac{1}{2} \end{matrix} \middle| p^2 z^{2q} \right)$$

Functions involving $\operatorname{erfi}(pz^q)$:

$$\operatorname{erfi}(pz^q) = \frac{pz^q}{\sqrt{\pi}} G_{1,2}^{1,1} \left(\begin{matrix} \frac{1}{2} \\ 0 \end{matrix} \middle| -p^2 z^{2q} \right)$$

Functions involving $E_a(pz^q)$:

$$E_a(pz^q) = G_{1,2}^{2,0} \left(\begin{matrix} a \\ a-1, 0 \end{matrix} \middle| pz^q \right)$$

Functions involving $C(pz^q)$:

$$C(pz^q) = \frac{1}{2} G_{1,3}^{1,1} \left(\frac{1}{\frac{1}{4}} \quad 0, \frac{3}{4} \middle| \frac{\pi^2 p^4}{16} z^{4q} \right)$$

Functions involving $S(pz^q)$:

$$S(pz^q) = \frac{1}{2}G_{1,3}^{1,1}\left(\frac{1}{\frac{3}{4}} \quad 0, \frac{1}{4} \middle| \frac{\pi^2 p^4}{16} z^{4q}\right)$$

Functions involving $\log(pz^q)$:

$$\log^n(pz^q) = \text{generated}$$

$$\begin{aligned}\log(a + pz^q) &= G_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(a) + G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(a) + G_{2,2}^{1,2} \left(\begin{matrix} 1, 1 \\ 1 \end{matrix} \middle| \frac{pz^q}{a} \right) \\ \log(|a - pz^q|) &= G_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(|a|) + G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) \log(|a|) + \pi G_{3,3}^{1,2} \left(\begin{matrix} 1, 1 \\ 1 \end{matrix} \middle| \frac{\frac{1}{2}}{0, \frac{1}{2}} \middle| \frac{pz^q}{a} \right)\end{aligned}$$

Functions involving $\sin(pz^q)$:

$$\sin(pz^q) = \sqrt{\pi} G_{0,2}^{1,0} \left(\frac{1}{2} \mid \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\sinh(pz^q)$:

$$\sinh(pz^q) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(\begin{matrix} 1 \\ \frac{1}{2}, 1, 0 \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\theta(-pz^q + 1)$, $\log(pz^q)$:

$$\log^n(pz^q)\theta(-pz^q+1)=\text{generated}$$

$$\log^n(pz^q)\theta(pz^q-1)=\text{generated}$$

5.5 Numerical evaluation

5.5.1 Floating-point numbers

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

[illegible]

As the last example shows, some Python floats are only accurate to about 15 digits as inputs, while others (those that have a denominator that is a power of 2, like $0.125 = 1/8$) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:

The exception indicates that `N` failed to achieve full accuracy. 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.

Also, you can set `strict` keyword argument to `False` to obtain imprecise answer instead of exception. For example, 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, strict=False)
0.e-126
>>> N(f, maxn=1000, strict=False)
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)
3.0000000000000000
```

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
```

5.5.3 Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```
>>> 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()
Traceback (most recent call last):
PrecisionExhausted: ...
```

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()
Traceback (most recent call last):
PrecisionExhausted: ...
```

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(pretty_print=True, wrap_line=False, no_global=True)
>>> Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)

$$\int_1^{\infty} \frac{\sin(x)}{x^2} dx$$

>>> N(, 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:

```
>>> Sum(1/k**2, (k, 1, oo)).evalf(strict=False)
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 π can be summed to 10,000 digits in a fraction of a second with a simple command:

```
>>> f = factorial
>>> R = 9801/sqrt(8)/Sum(f(4*n)*(1103+26390*n)/f(n)**4/396**(4*n),
... (n, 0, oo))
>>> N(R, 10000, strict=False)
3.141592653589793238462643383279502884197169399375105820974944592307...
```

5.5.4 Numerical simplification

The function `nsimplify` attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by `nsimplify` is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, `nsimplify` can be passed a list of constants to include (e.g. π) and a minimum numerical tolerance. Here are some elementary examples:

```
>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355/113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
>>> nsimplify(2.0**(1/3.), tolerance=0.001)
635/504
>>> nsimplify(2.0**(1/3.), tolerance=0.001, full=True)
3
sqrt(2)
```

Here are several more advanced examples:

```
>>> nsimplify(Float('0.130198866629986772369127970337', 30), [pi, E])
1
5*pi
----- + 2*e
7
>>> nsimplify(cos(atan('1/3')))
3*sqrt(10)
-----
10
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*phi
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49 + 8*i
-----
17 + 17i
```

(continues on next page)

(continued from previous page)

```
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1 - i*sqrt(5) + 1
2 - 10 4
>>> nsimplify(I**I, [pi])
- pi
2
e
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
pi^2
6
>>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])
sqrt(2)*pi
```

5.5.5 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

```
for(int i = 0; i < n; i++)
{
    temp[i] = sin(x[i]);
}
for(int i = 0; i < n; i++)
{
    y[i] = temp[i] / x[i];
}
```

This is slightly sub-optimal because

1. We allocate an extra `temp` array
2. We walk over `x` memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

```
for(int i = 0; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}
```

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, Diofant 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 Diofant user wishing to make use of this code generation should call the `ufuncify` function

```
>>> expr = sin(x)/x
```

```
>>> from diofant.utilities.autowrap import ufuncify
>>> f = ufuncify((x,), expr)
```

This function `f` consumes and returns a NumPy array. Generally `ufuncify` performs at least as well as `lambdify`. If the expression is complicated then `ufuncify` often significantly outperforms the NumPy backed solution. Jensen has a good [blog post](#) on this topic.

DEVELOPMENT

Warning: If you are new to Diofant, start with the [Tutorial](#) (page 3). If you are willing to contribute - it's assumed you know the Python programming language and the Git Version Control System.

This project adheres to [No Code of Conduct](#). Contributions will be judged by their technical merit.

6.1 Reporting Issues

When opening a new issue, please take the following steps:

1. Please search [GitHub issues](#) to avoid duplicate reports.
2. If possible, try updating to master and reproducing your issue.
3. Try to include a minimal code example that demonstrates the problem.
4. Include any relevant details of your local setup (Diofant version, Python version, installed libraries).

Note: Please avoid changing your messages on the GitHub, unless you want fix a typo and so on. Just expand your comment or add a new one.

6.2 Contributing Code

All work should be submitted via [Pull Requests \(PR\)](#).

1. PR can be submitted as soon as there is code worth discussing. Please make a draft PR, if one is not intended to be merged in its present shape even if all checks pass.
2. Please put your work on the branch of your fork, not in the master branch. PR should generally be made against master.
3. One logical change per commit. Make good commit messages: short (≤ 78 characters) one-line summary, then newline followed by verbose description of your changes. Please [mention closed issues](#) with commit message.

4. Please conform to [PEP 8](#) and [PEP 257](#); run:

```
flake518
```

to check formatting.

5. PR should include tests:

1. Bugfixes should include regression tests (named as `test_issue_123` for Diofant issues and as `test_sympyissue_123` for SymPy issues).
2. All new functionality should be tested, every new line should be covered by tests. Please use in tests only public interfaces. Regression tests are not accounted in the coverage statistics.
3. Optionally, provide doctests to illustrate usage. But keep in mind, doctests are not tests. Think of them as examples that happen to be tested.

6. It's good idea to be sure that **all** existing tests pass and you don't break anything, so please run:

```
pytest
```

To check also doctests, run:

```
pytest --doctest-modules
```

7. Please also check for potential flaws in your Python code with:

```
pylint diofant
```

and do type checking:

```
mypy diofant
```

8. If your change affects documentation, please build it by:

```
sphinx-build -W -b html docs build/sphinx/html
```

and check that it looks as expected.

6.3 Rosetta Stone

The Diofant project is a [SymPy](#)'s fork, so it could be handy to collect here some facts about SymPy and explain historical code conventions.

First, the SymPy project was hosted in SVN repository on the Google Code and our master branch include only commits, that added after moving project on the Github. But it's not a problem for us - we keep old history on the branch `sympy-svn-history`. Also, you can see this history as part of master's, if you [clone our repo](#) (page 1) and simply do this:

```
git fetch origin 'refs/replace/*:refs/replace/*'
```

Please note, that we have dozens of references to SymPy issues in our codebase. Such reference must be either a direct URL of the issue, or a fully qualified reference in the Github format, like `sympy/sympy#123`. Unqualified references like `#123` or `issue 123` — are reserved for our [Github issues](#).

However, in the old Git history, before commit [cbdd072](#), please expect that #123, issue #123 or issue 123 — are references to the SymPy's issues. The whole story is a little worse, because before commit [6f68fa1](#) - such unqualified references assume issues on the Google Code, not Github, unless other clearly stated. SymPy issues from the Google Code were moved to the Github in March 2014 (see [sympy/sympy#7235](#)). Transferred issue numbers were shifted by 3099. I.e. issue 123 in the git history before [6f68fa1](#) - does mean issue [sympy/sympy#3222](#) on Github.

6.4 Versioning

We use standard [Semantic Versioning](#) numbering scheme, but adopt [PEP 440](#) for alpha ("aN" suffix), beta ("bN") and development (".devN") releases.

6.5 Release Procedure

To release a new version, tag the latest commit in the master branch and publish this release tag:

```
git pull
git tag -s vX.Y.Z
git push origin vX.Y.Z
```


ABOUT

This Python library for symbolic mathematics is a fork of the [SymPy](#), started by Sergey B Kirpichev, last regular [SymPy](#)'s commit is [cbdd072](#) (22 Feb 2015). The git history goes back to 2007.

The project was named after the Diophantus of Alexandria. His "Arithmetica" is one of the earliest known texts that use symbols in equations. "Diofant" is a transliteration of Диофант, from Russian.

7.1 License

Unless stated otherwise, all files in the [Diofant](#) project are licensed using the new BSD license:

Copyright (c) 2006-2022 SymPy Development Team, 2013-2024 Sergey B Kirpichev

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder 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.

RELEASE NOTES

This section documents the changes that have been made in various versions of Diofant. Users should consult these pages to learn about new features, bug fixes and backwards incompatibilities. Developers should summarize the development work they do here in a user friendly format.

8.1 Diofant 0.15

Not Released Yet

8.1.1 New features

- New configuration option (`MAX_INTEGER_NBITS`) to control the maximal size of evaluated integers, see [#1327](#).
- Added `eliminate()` (page 529) to eliminate symbols from the equations, see [#1331](#).

8.1.2 Major changes

8.1.3 Compatibility breaks

- Removed `itermonomials()` and `topological_sort()` functions, see [#1321](#) and [#1322](#).
- Removed `Float.num` property, use `mpmath.mpmathify()`, see [#1323](#).
- Removed support for CPython 3.10, see [#1344](#).
- Removed `rcall()` method of *Basic* (page 46), see [#1346](#).
- Removed method argument of `jn_zeros()` (page 355), see [#1352](#).
- Removed `sstrrepr()` function, see [#1362](#).
- Removed support for ASCII pretty-printing and `pprint_use_unicode()` function, see [#1369](#).
- Removed `bottom_up()`, `has_variety()` and `has_dups()` functions, see [#1380](#).
- Removed `diofant.tensor.tensor` module, see [#1380](#).
- Removed `symarray()` function, see [#1383](#).

8.1.4 Minor changes

- Support CPython 3.12, see [#1325](#).

8.1.5 Developer changes

8.1.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These SymPy issues also were addressed:

- [sympy/sympy#25142](#): incorrect simplification of a complex relational
- [sympy/sympy#19813](#): logcombine hangs
- [sympy/sympy#22450](#): Rational raised to the big power hangs
- [sympy/sympy#25165](#): Series expansion not working
- [sympy/sympy#25197](#): Simple exponential integral error in an otherwise case
- [sympy/sympy#23399](#): Simplifying equation with function seemingly gets stuck
- [sympy/sympy#20427](#): Result from `clear_denoms()` prints like zero poly but behaves wierdly (due to unstripped DMP)
- [sympy/sympy#2720](#) `eliminate()`
- [sympy/sympy#16951](#): `integrate(sqrt(2*m*(E - x)), x)`
- [sympy/sympy#25341](#): CoercionFailed on eq: $2\sqrt{x}/(x + 1)^2 - 1/(\sqrt{x}(x + 1)) - 1/(4x^{3/2})/(x + 1) = 0$
- [sympy/sympy#20327](#): Finite Field coercion fails from Rational type
- [sympy/sympy#25406](#): Resultant of Polynomials Returns Wrong Output
- [sympy/sympy#25451](#): Incorrect simplification when mixing basic logical operators and equality
- [sympy/sympy#25496](#): Privileging `expr.__class__` over `expr.func` for reconstruction
- [sympy/sympy#25521](#): `integrate` raises `HeuristicGCDFailed`
- [sympy/sympy#25520](#): `RecursionError` in `inverse_laplace_transform`
- [sympy/sympy#25399](#): Cannot use `typing.Generic[T]` with `Symbol`
- [sympy/sympy#25582](#): Incorrect limit for `atan`
- [sympy/sympy#25592](#): `factor_list` sometimes generates `PolificationFailed` errors with algebraic extensions
- [sympy/sympy#25590](#): `simplify` produces wrong answer with non-commuting symbols
- [sympy/sympy#25572](#): `simplify` reorders noncommutative factors
- [sympy/sympy#25603](#): Simplifying And boolean operation removes a condition
- [sympy/sympy#25612](#): Lack of `is_real` attribute for `Mul` class
- [sympy/sympy#25624](#): `lcm(-1,1)` and `lcm(Poly(-1,x), Poly(1,x))` gives different output
- [sympy/sympy#25627](#): `solve` does not take `positive=True` into account

- [sympy/sympy#25681](#): Issues with limits while using abs function
- [sympy/sympy#25682](#): Branches for series expansions involving the abs function is not handled correctly
- [sympy/sympy#25679](#): hypersimp does not work correctly
- [sympy/sympy#25698](#): `n=6000002; int(n*(log(n) + log(log(n))))` takes more than 200 s to compute on [sympy.live.org](#)
- [sympy/sympy#25701](#): `TypeError` on `Eq(2*sign(x + 3)/(5*Abs(x + 3)**(3/5)), 0)`
- [sympy/sympy#25723](#): GCD missing polynomial factor
- [sympy/sympy#25738](#): Incorrect result of `reduce_inequalities` involving `pi` and `abs`
- [sympy/sympy#25697](#): can not reduce log inequalities
- [sympy/sympy#25806](#): Integrate a simple function
- [sympy/sympy#25833](#): Limit at infinity of `arctan(expression that goes to infinity)` erroneously gives `NaN` or it doesn't compute.
- [sympy/sympy#25882](#): `IndexError` when run `classify_ode`
- [sympy/sympy#25885](#): Wrong result for a limit
- [sympy/sympy#25886](#): `CeortionError` in `integrate()`
- [sympy/sympy#25896](#): `ratint(e,x).diff().equals(e)` is not always `True` (terms lost)
- [sympy/sympy#25899](#): surprising error message with `Poly('a-a')`
- [sympy/sympy#23843](#): Asymptotic series for `atan/acot` functions shifted to their branch cuts gives wrong answer
- [sympy/sympy#25965](#): `ceiling(CRootOf())` not implemented, leads to exception in `Range`
- [sympy/sympy#25983](#): Incorrect result of `reduce_inequalities`
- [sympy/sympy#25991](#): Inconsistencies in `as_leading_term`, `Series Expansion`, and `Limit Computations for Expressions Involving Square Roots`
- [sympy/sympy#26071](#): Definite integral error
- [sympy/sympy#26119](#): `Lambdify` crashes on empty tuple
- [sympy/sympy#26178](#): Wrong result of `sqf_list` for `PolyElement` with excluded symbols
- [sympy/sympy#26250](#): Incorrect limit involving elliptic functions
- [sympy/sympy#25786](#): Wrong result for a simple integral
- [sympy/sympy#26343](#): `TypeError: Invalid NaN Comparison` using `dsolve` for ODE with `ics={v(0) : 0}`
- [sympy/sympy#26313](#): Error result for limit of a piecewise
- [sympy/sympy#26477](#): Error in integral result using `hyper`
- [sympy/sympy#26497](#): `factor` produces wrong output
- [sympy/sympy#26501](#): `TypeError: '>' not supported between instances of 'Poly' and 'int'` calling `integrate` in `sympy 1.12`
- [sympy/sympy#26503](#): `TypeError: Invalid NaN comparison` calling `integrate` in `sympy 1.12`
- [sympy/sympy#26504](#): `IndexError: Index out of range: calling integrate` in `sympy 1.12`

- [sympy/sympy#26506](#): RecursionError: maximum recursion depth exceeded in comparison calling integrate in sympy 1.12
- [sympy/sympy#26513](#): Wrong limit result for $\text{Abs}((-n/(n+1))^{**n})$
- [sympy/sympy#26502](#): lots of PolynomialError contains an element of the set of generators exceptions calling integrate in sympy 1.12
- [sympy/sympy#14069](#): Condition for TODO in zeta_functions.py is now satisfied
- [sympy/sympy#25931](#): Possible improvements in gruntz
- [sympy/sympy#26566](#): wrong result integrating $\sin(a*(x+\pi))^{**2}$
- [sympy/sympy#26571](#): `integrate((x**8+1)**(-1/2),x)` raises `_CoeffExpValueError`
- [sympy/sympy#26577](#): factor should use the same domain for numerator and denominator

8.2 Diofant 0.14

12 Apr 2023

8.2.1 New features

- Support calculating limits with *Piecewise* (page 297) functions and boolean expressions, see [#1214](#) and [#1218](#).
- Support directional limits on the complex plane, see [#1232](#).

8.2.2 Major changes

- Use recursive (former `poly()` function, without using `expand()` (page 129)) algorithm of creating polynomials from expressions, see [#1047](#).

8.2.3 Compatibility breaks

- Removed support for CPython 3.9, see [#1191](#) and [#1192](#).
- Removed `to_mpi()` method of *Interval* (page 572), see [#1194](#).
- Removed `poly()` function, use `as_poly()` (page 64) method to create a *Poly* (page 507) instance from *Expr* (page 57), see [#1047](#).
- Removed functions `bool_map()`, `POSform()` and `SOPform()`, see [04ea41a220](#) and [be319badf5](#).
- Changed semantics of the `dir` kwarg for the *Limit* (page 752), now `'+'` is -1, `'-'` is 1 and `'real'` is *Reals* (page 580), see [#1234](#) and [#1235](#).
- Removed `diofant.calculus.euler` and `diofant.calculus.finite_diff` modules, see [#1271](#).
- Removed `diofant.vector` module, see [#1274](#).
- Removed `diofant.diffgeom` module, see [#1281](#).

- Removed `diofant.stats` module, see [#1276](#).
- Removed `diofant.geometry` module and `line_integrate()` function, see [#1283](#).
- Removed `diofant.plotting` module, see [#1284](#).
- Removed unused `prefixes()`, `postfixes()`, `capture()` and `variations()` functions, see [#1282](#) and [#1290](#).
- Drop support for multivariate *Order* (page 754) notion, see [#1296](#).
- Removed `extract_leading_order()` method of *Add* (page 104), see [#1292](#).
- Removed `S.UniversalSet` singleton object and related class, see [#1308](#).
- Removed unused `slice()` method of the *Poly* (page 507), see [#1318](#).

8.2.4 Minor changes

- Support unevaluated *RootOf* (page 540)'s over finite fields, see [#1209](#).
- Provide default clause (condition *BooleanTrue* (page 417)) for *Piecewise* (page 297), see [#1215](#).
- Support limits for *RootSum* (page 541), see [#1268](#).
- Support `--unicode-identifiers` module option, which allows using any unicode identifiers in interactive sessions, see [#1314](#).

8.2.5 Developer changes

- Use `pyproject.toml` to keep project's metadata, see [#1226](#).
- Drop dependency on the *flake8-rst* and depend on the *flake518* instead, see [#1268](#).

8.2.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These SymPy issues also were addressed:

- [sympy/sympy#22487](#): [integrals] Wrong result for $\text{Integral}((\cos(x^2)-\cos(x))/x^2, (x, -\infty, \infty))$
- [sympy/sympy#22493](#): Series expansion introduces new variables
- [sympy/sympy#22558](#): Error in ODE-Solver-Documentation
- [sympy/sympy#22837](#): Solve simplest algebraic equations with dummy parameter
- [sympy/sympy#22836](#): Series: Possible improvements for Order of expressions involving factorials
- [sympy/sympy#22788](#): RecursionError for unevaluated expression in latex
- [sympy/sympy#22863](#): Hangs: $\text{integrate}((3x^3-x^2+2x-4)/\sqrt{x^2-3x+2}, (x, 0, 1))$
- [sympy/sympy#22862](#): Problem with separable differential equation
- [sympy/sympy#22893](#): 'limit' in combination with 'positive=True' gives wrong result

- [sympy/sympy#22878](#): RecursionError in trigsimp
- [sympy/sympy#22982](#): `limit((log(E + 1/x) - 1)**(1 - sqrt(E + 1/x)), x, oo)` returns 0 instead of oo
- [sympy/sympy#22986](#): `limit(acosh(1 + 1/x)*sqrt(x), x, oo)` is evaluated incorrectly.
- [sympy/sympy#14433](#): `x` not in `QQ.frac_field(1/x)`
- [sympy/sympy#23069](#): `integrate(r**4*sqrt(1 - r**2), (r, 0, 1))` gives incorrect result
- [sympy/sympy#19639](#): TypeError in integrate
- [sympy/sympy#23086](#): Incorrect result of simplify
- [sympy/sympy#23156](#): `sympy.Sum()` bug when summing up reciprocal of gamma
- [sympy/sympy#23174](#): Problem with `gf_edf_zassenhaus()`
- [sympy/sympy#21409](#): Printing of polynomial over FF
- [sympy/sympy#22673](#): Roots of a polynomial over a finite field computed regardless of specified polynomial domain
- [sympy/sympy#12531](#): `cancel` does not return expanded form
- [sympy/sympy#6322](#): `degree((x+1)**10000)` takes too long
- [sympy/sympy#22583](#): `is_polynomial` right for wrong reasons (and sometimes wrong)
- [sympy/sympy#23202](#): Dropping “all” `__ne__` methods?
- [sympy/sympy#23223](#): Wrong integration results of trigonometric functions
- [sympy/sympy#23224](#): Python code printer not respecting tuple with one element
- [sympy/sympy#23231](#): Sympy giving the wrong solution
- [sympy/sympy#14387](#): Tutorial on limits creates impression that they are two-sided by default
- [sympy/sympy#8166](#): Limit assumes function is continuous?
- [sympy/sympy#14502](#): Problem with limit including factorial.
- [sympy/sympy#18492](#): Limit of Piecewise function - NotImplementedError: Don't know how to calculate the mrv
- [sympy/sympy#23266](#): Regression(?) in 1.10 for limits
- [sympy/sympy#7391](#): Limits for expressions with undetermined functions give wrong results
- [sympy/sympy#23287](#): Regression in `is_integer` for Mul of Pow
- [sympy/sympy#11496](#): Wrong result in limit calculation of `limit(erfc(ln(1/x)),x,oo)?`
- [sympy/sympy#3663](#): series expansion of `acosh` and `acoth`
- [sympy/sympy#23299](#): Sympy is unable to integrate this
- [sympy/sympy#23319](#): testing limit of `n*tan(pi/n)` results in incorrect answer in 1.7rc1+
- [sympy/sympy#5539](#): Equal Integrals compare different when using different variables
- [sympy/sympy#23425](#): PolynomialError when I try to call `classify_ode`
- [sympy/sympy#23432](#): Series expansion around float fails with NotImplementedError
- [sympy/sympy#8433](#): limit involving error function returns bad result

- [sympy/sympy#13750](#): erf has wrong limit in $-\infty$
- [sympy/sympy#23497](#): `binomial(-1, -1)` returns 0, should return 1
- [sympy/sympy#23562](#): In new version of sympy, `dsolve` does not give a solution when another derivative is involved
- [sympy/sympy#23585](#): `FiniteSet` documentation inconsistent with usage in sympy
- [sympy/sympy#23596](#): Integral of real function has complex result
- [sympy/sympy#23605](#): Inefficiency in the Integrator with a Rational Expression
- [sympy/sympy#23637](#): Missing solutions from polynomial system (various solvers)
- [sympy/sympy#23479](#): Sparse poly gcd fails with `HeuristicGCDFailed('no luck')`
- [sympy/sympy#22605](#): Incorrect result from `minpoly(cos(pi/9))`
- [sympy/sympy#23677](#): `minimal_polynomial` fails for very complicated algebraic number
- [sympy/sympy#23836](#): Incorrect results for limits of `Piecewise` at discontinuity
- [sympy/sympy#23845](#): `Gruntz` should have been free of `_w`, value error, recursion error
- [sympy/sympy#23855](#): `linsolve` gives odd result if symbols are duplicated
- [sympy/sympy#24067](#): incorrect limit in simple parametric rational polynomial
- [sympy/sympy#24127](#): Error on all limits with `Piecewise`
- [sympy/sympy#23702](#): Cannot specify ODE initial conditions as just `f(0)`
- [sympy/sympy#23707](#): `AttributeError` in integral
- [sympy/sympy#24210](#): Error on limits regarding terms like $(1+u)^v$.
- [sympy/sympy#24225](#): Multivariable limit should be undefined, but gives unity.
- [sympy/sympy#24266](#): Changed behaviour of `series()` involving `exp`, `I`
- [sympy/sympy#24331](#): Limit of $\log(z)$ as z goes to 0 with z complex returns `'-oo'` instead of `'zoo'`
- [sympy/sympy#23766](#): `Factor` hangs on exponential functions with base `e`
- [sympy/sympy#24360](#): Remove usage of `numpy.distutils` in `autowrap` module
- [sympy/sympy#24346](#): `factor` with `extension=True` fails for rational expression
- [sympy/sympy#20913](#): `Poly(x + 9671406556917067856609794, x).real_roots()` is slow
- [sympy/sympy#24386](#): `sympy.limit` yields wrong limit in sigmoidal expression
- [sympy/sympy#24390](#): Incorrectly evaluated expression
- [sympy/sympy#24461](#): `sympy.polys.polyerrors.HeuristicGCDFailed: no luck` - when multiplying two `Polys`
- [sympy/sympy#24543](#): Rational calc value error
- [sympy/sympy#6326](#): `PolynomialRing` should not derive from `CharacteristicZero`
- [sympy/sympy#24684](#): Unable to evaluate `erfcinv`
- [sympy/sympy#6822](#): `Multivariate Order()`
- [sympy/sympy#24477](#): `Expand` before `integrate` gives different results with big `O`
- [sympy/sympy#24928](#): `simplify(asinh(2)-oo)->0`

- [sympy/sympy#24948](#): `.is_positive` returns `None` when it should be `False`
- [sympy/sympy#24957](#): Timeout for `dsolve((2x^3+3y)+(3x+y-1)y'=0)`
- [sympy/sympy#24955](#): Timeout for `dsolve(x^2*y'-y^2*y'+2*x*y=0)`
- [sympy/sympy#22943](#): `RootOf` for polynomials with irrational algebraic coefficients

8.3 Diofant 0.13

7 Nov 2021

8.3.1 New features

- Support square-free factorization of multivariate polynomials over finite fields (with adaptation of Musser's algorithm), see [#1132](#).

8.3.2 Major changes

- Support calling from the command-line as `python -m diofant`, see [#853](#). Thanks to André Roberge.

8.3.3 Compatibility breaks

- Removed `n()` method from *EvalfMixin* (page 138), see [#1114](#).
- Former submodule `diofant.polys.polyconfig` now is *diofant.config* (page 41), see [#1115](#).
- Drop support for `DIOFANT_DEBUG` environment variable, see [#1115](#).
- Drop support for CPython 3.7 and 3.8, see [#1118](#) and [5cae972](#).
- Renamed `Ring` as *CommutativeRing* (page 428), see [#1123](#).
- Removed support for Python 3.7 and 3.8, see [#1118](#) and [#1124](#).
- `FiniteRing` renamed to *IntegerModRing* (page 429), see [#1124](#).
- Removed `igcd()`, `ilcm()` and `prod()` functions, see [#1125](#).
- Changed the *Derivative* (page 122) (and similary *diff()* (page 125)) syntax to `Derivative(foo, (x, 2))` from `Derivative(foo, x, 2)`, see [#1131](#).
- Removed `prem()` function, see [#1140](#).
- Removed `lseries()` method of *Expr* (page 57), use *series()* (page 79) with `n=None`, see [#1146](#).

8.3.4 Minor changes

- Protect hashed *PolyElement* (page 758)'s from modifications, see [#1033](#).
- Add gaussian rationals as an exact domain, associated with *ComplexField* (page 431), see [#1138](#).
- Support *tan* (page 282) in *minimal_polynomial()* (page 539), see [#1159](#).
- 100% test coverage for plotting module, see [#1175](#).
- Support CPython 3.10, see [#1162](#).

8.3.5 Developer changes

- Turn on type checking for the whole codebase, see [#1114](#).
- Don't include regression tests in the coverage statistics, see [#1060](#).

8.3.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These SymPy issues also were addressed:

- [sympy/sympy#20861](#): `reduce_inequalities()` gives impossible answer
- [sympy/sympy#20874](#): Port the PRS algorithm to the sparse polynomial implementation
- [sympy/sympy#20902](#): Incorrect inequality solving: False returned instead of answer
- [sympy/sympy#20941](#): Fails to Solve Definite Integral
- [sympy/sympy#20973](#): `cancel` raises `PolynomialError` for `exp(1+O(x))`
- [sympy/sympy#20985](#): `TypeError`s appearing for simple polynomial manipulations (did not happen in v1.6.1)
- [sympy/sympy#21031](#): Limit of “`limit (((1+x)**(1/x)-(1+2*x)**(1/(2*x)))/asin(x),x,0)`” is wrong with v1.7.1
- [sympy/sympy#21034](#): (Integration) regressions?
- [sympy/sympy#21038](#): Incorrect computation of a basic limit, regression from 1.6.2 to 1.7.1
- [sympy/sympy#21041](#): integrate error
- [sympy/sympy#21063](#): Wrong value of improper integral when using unevaluated `-oo` as boundary
- [sympy/sympy#21075](#): Order term being added to exact expansion
- [sympy/sympy#21091](#): Invalid comparison of non-real when using `integrate()`
- [sympy/sympy#19590](#): `Poly.diff()` doesn't support higher order derivatives
- [sympy/sympy#21121](#): Same symbols created in different processes are not resolved as being equal
- [sympy/sympy#21107](#): `S.Infinity.is_nonzero` returns False
- [sympy/sympy#21132](#): Integral with parametres: wrong and too long result

- [sympy/sympy#21180](#): Bug: `sympy.factor` doesn't work for Poly !!!
- [sympy/sympy#21167](#): Empty list of solutions returned for equation with cubic roots
- [sympy/sympy#21029](#): Continuous limits involving division by x
- [sympy/sympy#20697](#): Series is not simplified to final answer in output in sympy 1.7.1
- [sympy/sympy#20578](#): A strange behavior of limit function
- [sympy/sympy#20444](#): Leading Term with log
- [sympy/sympy#19453](#): Limit changes from simplification of original expression
- [sympy/sympy#19442](#): Non-existent bi-directional limit gives `ValueError`
- [sympy/sympy#11667](#): `limit(1/x, x, 0) == oo ??`
- [sympy/sympy#21202](#): `laplace_transform(cosh(2*x), x, s)` raises `RecursionError`
- [sympy/sympy#21227](#): Nested logarithms add unnecessary order term to series expansions
- [sympy/sympy#21263](#): Solutions of cubic equation
- [sympy/sympy#21334](#): `RecursionError` while calculating leading term
- [sympy/sympy#21342](#): `1/(exp(it) - 2)` integrates wrong
- [sympy/sympy#21319](#): Primitive part of zero polynomial
- [sympy/sympy#21341](#): Issues with continued fraction for real roots of cubic polynomials
- [sympy/sympy#21024](#): `sympy.polys.polyerrors.CoercionFailed` integration regressions?
- [sympy/sympy#21396](#): `Pow.as_base_exp` inconsistent with `I.as_base_exp`
- [sympy/sympy#21410](#): Polynomial power raises `KeyError`
- [sympy/sympy#21437](#): `log(Abs)`
- [sympy/sympy#21460](#): Polynomial GCD result is different for dense trivial polynomial
- [sympy/sympy#21466](#): Regression for match for differential binomial expression
- [sympy/sympy#21166](#): Wrong integration result involving square root of absolute value
- [sympy/sympy#21486](#): `expand_func(besselj(oo, x)) -> RecursionError`
- [sympy/sympy#21530](#): Incorrect limit
- [sympy/sympy#21549](#): Bug: `integrate(x*sqrt(abs(x)), (x, -1, 0))` returns wrong result
- [sympy/sympy#21557](#): Summation of geometric series with non-real exponent does not evaluate
- [sympy/sympy#21550](#): Bug: limit returns wrong result for rational function
- [sympy/sympy#21177](#): Incorrect residue for `cot(pi*x)/(x**2 - 3*x + 3)`
- [sympy/sympy#21245](#): laurent series Fibonacci generating fuction
- [sympy/sympy#11833](#): error in limit involving exp, sinh and an assumption (maybe related to caching)
- [sympy/sympy#9127](#): `nttheory.AskEvenHandler.Mul` is order-dependent
- [sympy/sympy#21606](#): Notimplemented in simple limit
- [sympy/sympy#21641](#): Simplify hangs

- [sympy/sympy#21651](#): doit() method *sometimes* ignores floor and ceiling within Sum
- [sympy/sympy#20461](#): Eq(Product(4*n**2/(4*n**2 - 1), (n, 1, oo)), pi/2) incorrectly gives False
- [sympy/sympy#13029](#): with gens, time taken for sqf increases orders of magnitude faster than factor as input size increases
- [sympy/sympy#21711](#): odd result for integrate(sqrt(1 - (x-1)*(x-1)), (x, 0, 1))
- [sympy/sympy#21721](#): Bug in integration solver
- [sympy/sympy#21716](#): isympy -c python tab triggered auto completion not working
- [sympy/sympy#21741](#): integrate() does not work with multivariable function that is solved by simple substitution. DomainError: there is no ring associated with CC
- [sympy/sympy#21756](#): Incorrect limit with ratio of complex exponentials
- [sympy/sympy#21760](#): Poly div is slow
- [sympy/sympy#21761](#): sympy.polys.polyerrors.NotAlgebraic Exception
- [sympy/sympy#21430](#): minpoly raises 'NotAlgebraic' for tan(13*pi/45)
- [sympy/sympy#21766](#): solve breaks on certain repeated inputs
- [sympy/sympy#21773](#): TypeError multiplying Subs expressions
- [sympy/sympy#21785](#): Limit gives TypeError from as_leading_term
- [sympy/sympy#21812](#): LambertW displaying in jupyter lab
- [sympy/sympy#21814](#): Printing of unevaluated Mul needs brackets
- [sympy/sympy#21176](#): Incorrect residue of $x^2 \cot(\pi x) / (x^4 + 1)$
- [sympy/sympy#21852](#): simple quadratic not solving
- [sympy/sympy#21859](#): AttributeError: 'mpz' object has no attribute 'denominator' with sp.series()
- [sympy/sympy#21882](#): Incorrect solutions given by solve
- [sympy/sympy#21890](#): RecursionError and TypeError in nonlinsolve
- [sympy/sympy#21888](#): TypeError raised for evalf containing summations
- [sympy/sympy#5822](#): What should summation() do with non-integer limits?
- [sympy/sympy#19745](#): Weird value of a sum
- [sympy/sympy#9358](#): summation: Wrong out for non-integral range
- [sympy/sympy#21905](#): raise NotImplementedError("Equation not in exact domain. Try converting to rational") Error
- [sympy/sympy#21938](#): Series raises an error at infinity for an example which can be solved by aseries
- [sympy/sympy#21984](#): ValueError: list.remove(x): x not in list occurs in nonlinsolve
- [sympy/sympy#21999](#): detection of infinite solution request
- [sympy/sympy#22020](#): Comparing two operations that contain log sometimes leads to TypeError exception
- [sympy/sympy#22051](#): Nonlinsolve incorrect result

- [sympy/sympy#22058](#): Regression in solveset for quadratic with symbolic coefficients
- [sympy/sympy#22073](#): Interval with oo
- [sympy/sympy#22093](#): sympy.polys.polyerrors.HeuristicGCDFailed: no luck
- [sympy/sympy#22155](#): Problem with solving simple separable ODE
- [sympy/sympy#22220](#): Bug in the evaluation of a log limit
- [sympy/sympy#22248](#): solve running forever
- [sympy/sympy#22294](#): Bernoulli differential equation
- [sympy/sympy#22322](#): 'abs' is not parsed correctly
- [sympy/sympy#22334](#): Wrong answer returned while calculating limit for different arrangements of the same expression
- [sympy/sympy#22400](#): Minpoly doesn't terminate
- [sympy/sympy#22435](#): sympy integration error

8.4 Diofant 0.12

18 Jan 2021

8.4.1 New features

- Support modular exponentiation of *PolyElement* (page 758)'s, see [#1032](#).
- *reduce_inequalities()* (page 610) support solving linear inequalities with Fourier-Motzkin elimination algorithm, see [#1063](#).
- Added class *FiniteRing* for modular integers, see [#876](#).
- Implemented *compose()* (page 786) for functional composition in the fields of fractions, see [#1100](#).

8.4.2 Major changes

- Module *sqfreetools* (page 785) was ported to use sparse polynomial representation, see [#1009](#).
- Module *factortools* (page 767) was ported to use sparse polynomial representation, see [#1015](#), [#1018](#), [#1019](#), [#1020](#) and [#1021](#).
- Module *rootisolation* (page 784) was ported to use sparse polynomial representation, finally the dense representation is used nowhere, see [#1030](#), [#1031](#) and [#1035](#).
- *reduce_inequalities()* (page 610) uses *ExtendedReals* (page 580) subsets to solve inequalities, see [#1067](#) and [#1092](#).
- Added new algorithm for factorization of multivariate polynomials over *AlgebraicField* (page 429)'s (uses Hensel lifting), see [#876](#). Thanks to Katja Sophie Hotz. Thanks to Kalevi Suominen for help with review.

8.4.3 Compatibility breaks

- Removed `vring()` and `vfield()` functions, see [#1016](#).
- Drop support for `from_list()` initialization for multivariate polynomials, see [#1035](#).
- Drop `to_dense()`, `tail_degrees()`, `almosteq()` and `degree_list()` methods and `is_monic`, `is_primitive` attributes of *PolyElement* (page 758), see [#1035](#), [#1036](#) and [#1051](#).
- Drop `is_monic`, `is_primitive`, `zero`, `one` and `unit` attributes and `degree_list()` method of *Poly* (page 507), see [#1036](#), [#1039](#) and [#1051](#).
- Drop `sring()`, `poly_from_expr()`, `gcd_list()` and `lcm_list()` functions, see [#1037](#), [#1057](#) and [#1086](#).
- Functions and classes of the *polytools* (page 505) module do not support anymore iterables as polynomial generator, see [#1039](#).
- Drop unused functions `dispersion()`, `dispersionset()` and `degree_list()`, see [#1051](#) and [#1053](#).
- Drop rich comparison methods from the *FracElement* (page 786), see [#1101](#).
- `from_list()` (page 515) support now ascending order of coefficients (i.e., the leading coefficient of univariate polynomial is coming last), see [#1103](#).
- Removed support for 3D geometry in the geometry module and `Point.__getitem__()` method, see [#1105](#).
- Drop `coeff()`, `coeffs()`, `monoms()`, `terms()` and `deflate()` methods of *PolyElement* (page 758), use dictionary indexing, see [#1108](#).

8.4.4 Minor changes

- Special case univariate polynomials with *UnivarPolynomialRing* (page 430) and *UnivarPolyElement* (page 763), see [#1024](#).
- Implement `is_primitive` (page 432), see [#1035](#).
- Add *ExtendedReals* (page 580) singleton, see [#1067](#).
- 100% test coverage for geometry module, see [#1105](#). Overall test coverage is around 98%.

8.4.5 Developer changes

- Depend on `flake8-sfs`, see [#983](#).
- Depend on `mypy`, see [#1046](#).
- Drop dependency on `strategies`, see [#1074](#).

8.4.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These Sympy issues also were addressed:

- [sympy/sympy#19630](#) rsolve gives None for linear homogeneous recurrence relation
- [sympy/sympy#19076](#) modular exponentiation of poly
- [sympy/sympy#19670](#) Poly(E**100000000) is slow to create
- [sympy/sympy#19755](#) poly gives coercion error when integers and rationals are mixed
- [sympy/sympy#19760](#) minimal_polynomial using Groebner basis can give wrong result
- [sympy/sympy#19770](#) Limit involving cosine
- [sympy/sympy#19766](#) Incorrect limit
- [sympy/sympy#19774](#) evalf() doesn't evaluate terms in an exponential
- [sympy/sympy#19988](#) Float loses precision after being pickled
- [sympy/sympy#14874](#) Limit $x \rightarrow \infty$ for bessellk
- [sympy/sympy#19991](#) Wrong result from floor().evalf()
- [sympy/sympy#10666](#) resultant misses the sign
- [sympy/sympy#20163](#) Apart hangs with extension=[sqrt(3), I]
- [sympy/sympy#9479](#) Cannot solve multivariate inequalities
- [sympy/sympy#20365](#) Limit Bug
- [sympy/sympy#20360](#) Incorrect definite integration of simple exponential involving pi
- [sympy/sympy#20389](#) TypeError: Argument of Integer should be of numeric type, got -oo
- [sympy/sympy#20391](#) Linear programming with simplex method
- [sympy/sympy#19161](#) When applying simplify on a Poly it fails
- [sympy/sympy#20397](#) bug in dividing polynomials by module
- [sympy/sympy#19196](#) Slow f.factor_list
- [sympy/sympy#20491](#) Inconsistencies in pretty printing in a notebook
- [sympy/sympy#20490](#) LaTeX printing of negative constant PolyElement
- [sympy/sympy#20484](#) Need more utility for polynomial substitution
- [sympy/sympy#20485](#) Rational powers for non-monomial PolyElement
- [sympy/sympy#20487](#) LaTeX printing errors for puiseux polynomial
- [sympy/sympy#20610](#) Solve: GeneratorsNeeded with system involving constant equation
- [sympy/sympy#20617](#) Complex exponentials are not recognized by domains
- [sympy/sympy#20640](#) Multivariate polynomial division
- [sympy/sympy#20704](#) Limit not terminating

8.5 Diofant 0.11

22 Apr 2020

8.5.1 New features

- Added `discrete_log()` (page 252) to compute discrete logarithms, see [#785](#). Thanks to Gabriel Orisaka.
- Support inhomogenous case for systems of linear ODEs with constant coefficients, see [#919](#).
- Support domains pickling, see [#972](#).

8.5.2 Major changes

- `Poly` (page 507) now use sparse polynomial representation (via `PolyElement` (page 758)) internally, see [#795](#).
- `rsolve()` (page 684) now return `list` of `dict`'s, see [#940](#).
- `solve()` (page 606) now return all solutions for equations, involving surds, see [#910](#).
- Module `galoistools` was adapted to use `FiniteField` (page 429)'s and usual conventions for low-level methods of the `polys` (page 505) module, see [#957](#), [#971](#) and [#964](#). Polynomial factorization now works for univariate polynomials over any `FiniteField` (page 429)'s domain.
- Module `euclidtools` (page 764) was ported to use sparse polynomial representation, see [#994](#).

8.5.3 Compatibility breaks

- Removed support for Python 3.5 and 3.6, see [#775](#).
- `is_monomial` attribute of `Poly` (page 507) renamed to `is_term` (page 519), see [#780](#).
- Removed `log()` helper from `RationalField` (page 429), see [#787](#).
- Removed `seterr()` function, see [#794](#).
- Removed `DMP` class, see [#795](#).
- Removed `ring_series` module, see [#820](#).
- `Equality` (page 109) doesn't support single-argument call, see [#828](#).
- Removed `is_nonnegative()`, `is_nonpositive()` and `is_positive()` methods of `Domain` (page 427) subclasses, see [#834](#) and [#975](#).
- Change order of keyword arguments for `integrate()` (page 761), see [#834](#).
- Removed support for `dps=' '` in `Float` (page 85). Significant digits automatically counted for `int` and `str` inputs, see [#797](#).
- Removed `numer/denom` properties of `FracElement` (page 786), see [#851](#).
- Removed `is_hermitian/is_antihermitian` core properties, see [#873](#).

- Removed `print_python()` and `print_ccode()` functions, see #891.
- Reorder output for `jordan_form()` (page 462) and `jordan_cells()` (page 461), the last one is now optional, see #896.
- Removed `generate_oriented_forest()`, `kbins()` and `ibin()` functions, see #903.
- Removed support for numexpr module in `lambdify()` (page 745) and NumExprPrinter printer class, see #903.
- Removed DeferredVector class, see #905.
- Don't export too much from `solvers` (page 606) to the default namespace, keep only `solve()` (page 606), `rsolve()` (page 684) and `dsolve()` (page 635) functions, see #921.
- Make `rsolve()` (page 684)'s `init` parameter more compatible with `dsolve()` (page 635)'s one, e.g. drop accepting `init=[1, 2, 3]` and `init={0: 1, 1: 2, 2: 3}` forms, see #921.
- Removed `dict_merge()`, `generate_bell()` and `reshape()` functions, see #921.
- Removed `subs()` methods from `PolyElement` (page 758) and `FracElement` (page 786), see #967.
- `is_negative()` method of `Domain` (page 427) refactored to the `is_normal()` (page 428), see #977.
- Removed `algebraic_field()` method of `IntegerRing` (page 429), see #977.
- Removed `has_assoc_Field` property, `is_SymbolicDomain` property renamed to `is_ExpressionDomain` of `Domain` (page 427), see #977.
- `drop_to_ground()` method of `PolynomialRing` (page 429) renamed to `eject()` (page 429), see #977.
- Renamed option misspelled option `bareis` to `bareiss` in `det()` (page 449) and `wronskian()` (page 476), see #866.
- Removed `nth_power_roots_poly()`, `ground_roots()`, `refine_root()`, `intervals()` and `sturm()` functions and `nth_power_roots_poly()`, `ltrim()`, `ground_roots()`, `refine_root()`, `intervals()`, `max_norm()`, `l1_norm()` and `sturm()` methods of `Poly` (page 507), see #996.

8.5.4 Minor changes

- Support truncation for elements of `RealAlgebraicField` (page 429) to `int`, see #788.
- `Matrix` (page 432)'s and `Array` (page 700)'s support symbolic indexes, see #785. Thanks to Francesco Bonazzi.
- Added `AA_FACTOR_METHOD` configuration option to specify factorization algorithm for polynomials with algebraic coefficients, see #844.
- `CCodeGen` (page 717) got support for common subexpression replacement, see #893. Thanks to James Cotton.
- 100% test coverage for `utilities` (page 710) module.
- `rsolve()` (page 684) got `simplify` option to control default output simplification, see #921.
- Function `rsolve()` (page 684) got initial support for systems of equations, see #921.

- `minimal_polynomial()` (page 539) got support for `RootOf` (page 540) instances over algebraic number fields, see #927.
- The `CommutativeRing` (page 428) and all derived classes got `characteristic` (page 428) property, see #968.
- Correct wrong implementations of factorization algorithms over finite fields, see #968 and #964. Thanks to Kalevi Suominen for help with review.

8.5.5 Developer changes

- Depend on `sphinxcontrib-bibtex` to track the bibliography, see #766.
- Use Github Actions for CI, instead of the Travis CI, see #887.
- Depend on `flake8-rst` to test formatting of docstrings, see #928.
- Depend on `flake8-quotes`, see #982.

8.5.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These SymPy issues also were addressed:

- [sympy/sympy#15943](#) Wrong result from summation
- [sympy/sympy#12163](#) matematica code printer does not handle floats and derivatives correctly
- [sympy/sympy#11642](#) Geometric sum doesn't evaluate with float base
- [sympy/sympy#15984](#) Value error in limit
- [sympy/sympy#7337](#) Wrong integration result
- [sympy/sympy#11600](#) re and im should work for matrix expressions
- [sympy/sympy#16038](#) solve_poly_system works with integers but not floats
- [sympy/sympy#15553](#) rsolve can not solve this kind of recurrences
- [sympy/sympy#11581](#) conjugate of real expression should not change expression
- [sympy/sympy#11976](#) Typo in ellipse.py
- [sympy/sympy#11275](#) LaTeX printer inconsistent with pretty printer
- [sympy/sympy#11841](#) Function('gamma') pretty prints as Γ
- [sympy/sympy#11926](#) ccode does not accept user_functions for Max and Min
- [sympy/sympy#11855](#) DiracDelta function is zero for nonzero arguments
- [sympy/sympy#11955](#) diophantine gives wrong solution for $-4*x**2+4*x*y-y**2+2*x-3$
- [sympy/sympy#11502](#) Discrete logarithms
- [sympy/sympy#11435](#) str printing of logic expressions should use operators
- [sympy/sympy#12200](#) coeff docstring is wrong
- [sympy/sympy#9123](#) apart drops term

- [sympy/sympy#12177](#) Wrong result with apart Wrong Result
- [sympy/sympy#8129](#) The probability function does not handle expressions like $b \geq b$
- [sympy/sympy#9983](#) `Product(1 + 1/n**(S(2)/3), (n, 1, oo)).doit()` raise `RunTimeError`
- [sympy/sympy#11726](#) `pde_separate` does not allow expressions as input
- [sympy/sympy#11981](#) `powsimp()` fails with noncommutative variables
- [sympy/sympy#12092](#) `evalf` does not call `_imp_` recursively
- [sympy/sympy#10472](#) `pprint` should align the middle of the matrix to the baseline?
- [sympy/sympy#11959](#) diophantine gives wrong solution for $-4x^2 + 4xy - y^2 + 2x - 3$
- [sympy/sympy#11944](#) matrix `vstack/hstack` can fail with immutable matrix as first argument
- [sympy/sympy#11732](#) Fails operators between Interval and some S.Sets
- [sympy/sympy#12178](#) Empty intersection should be UniversalSet
- [sympy/sympy#10681](#) `TypeError: 'Float' object cannot be interpreted as an integer from integrate(r**2*(R**2-r**2)**0.5, r)`
- [sympy/sympy#11078](#) `TypeError: 'Float' object cannot be interpreted as an integer from integrate((6-x*x)**(1.5))`
- [sympy/sympy#11877](#) `integrate(log(0.5-x), (x, 0, 0.5))` wrongly produces imaginary part
- [sympy/sympy#7337](#) Wrong integration result
- [sympy/sympy#10211](#) `integrate((1/sqrt(((y-x)**2 + h**2))**3), (x,0,w), (y,0,w))` is wrong
- [sympy/sympy#11806](#) Incorrectly evaluating integral
- [sympy/sympy#12325](#) string formatting error in `dmp_integrate_in`
- [sympy/sympy#16222](#) `Poly(E**100000000)` is slow to create
- [sympy/sympy#15413](#) `rootof` fails for polynomial with irrational coefficients
- [sympy/sympy#16432](#) `a.is_even` does not imply `a.is_finite`
- [sympy/sympy#16431](#) `a.is_zero` is False does not imply `a.is_nonzero` is True
- [sympy/sympy#16530](#) `(1/x).is_real` should be None if x can be zero
- [sympy/sympy#16562](#) `Eq` with 1 argument is allowed?
- [sympy/sympy#16589](#) `roots` gives incorrect result
- [sympy/sympy#16714](#) `Limit ((n**(n+1) + (n+1)**n) / n**(n+1))**n` recursion error
- [sympy/sympy#16774](#) square proportion match has no result
- [sympy/sympy#17034](#) `isqrt` gives incorrect results
- [sympy/sympy#17044](#) `is_square` gives incorrect answers
- [sympy/sympy#10996](#) Bug in polynomial GCD computation
- [sympy/sympy#15282](#) Works too long on some limits with big powers
- [sympy/sympy#16722](#) `limit(binomial(n + z, n)*n**-z, n, oo)` gives different answers based on assumptions of n and z
- [sympy/sympy#15673](#) Wrong results. (Limit, Integral, sphere(Space polar coordinates))

- [sympy/sympy#17380](#) Incorrect results given by some limit expressions
- [sympy/sympy#17431](#) Wrong results. (Limit, factorial, Power)
- [sympy/sympy#17492](#) Add link to GitHub in the Sphinx documentation
- [sympy/sympy#17555](#) `(-x).is_extended_positive` fails for `extended_real` and `infinite`
- [sympy/sympy#17556](#) `Mul.is_imaginary` fails for infinite values
- [sympy/sympy#17453](#) `Pow._eval_is_error`
- [sympy/sympy#17719](#) `plot_implicit` error for `Xor`
- [sympy/sympy#12386](#) Latex printer for `MutableDenseNDimArray`, `MutableSparseNDimArray`
- [sympy/sympy#12369](#) Start using `spherical_jn` from SciPy
- [sympy/sympy#17792](#) Wrong limit
- [sympy/sympy#17789](#) Intermittent test failure in assumptions
- [sympy/sympy#17841](#) `integrate` throws error for rational functions involving `I`
- [sympy/sympy#17847](#) Wrong result for `as_leading_term()`
- [sympy/sympy#17982](#) Wrong result from `rsolve`
- [sympy/sympy#9244](#) `dsolve`: nonhomogeneous linear systems are not supported
- [sympy/sympy#15946](#) Matrix exponential for `dsolve`
- [sympy/sympy#16635](#) problem when using `dsolve()` to solve ordinary differential equations
- [sympy/sympy#14312](#) Incorrect solution of 3 by 3 linear ODE systems
- [sympy/sympy#8859](#) wrong result: `dsolve` for systems with forcings
- [sympy/sympy#9204](#) `dsolve` fails
- [sympy/sympy#14779](#) Spurious solutions when solving equation involving `Abs(x)/x`
- [sympy/sympy#18008](#) series does not give the same expansion depending on whether simple expression is simplified or not
- [sympy/sympy#8810](#) Poly keyword 'composite' is ignored when instantiating from Poly
- [sympy/sympy#18118](#) `limit(sign(sin(x)), x, 0, '+')` = 0 (which is wrong)
- [sympy/sympy#6599](#) limit of fraction with oscillating term in the numerator calculated incorrectly
- [sympy/sympy#18176](#) Incorrect value for `limit(x**n-x**(n-k),x,oo)` when `k` is a natural number
- [sympy/sympy#18306](#) `NotImplementedError` in `limit`
- [sympy/sympy#8695](#) `sqf` and `sqf_list` output is not consistent
- [sympy/sympy#18378](#) Invalid result in `Limit`
- [sympy/sympy#18384](#) `abs(sin(x)*cos(x))` integrates wrong
- [sympy/sympy#18399](#) Incorrect limit
- [sympy/sympy#18452](#) Infinite recursion while computing Limit of Expression in 1.5.1
- [sympy/sympy#18470](#) `nan**0` returns 1 instead of `nan`

- [sympy/sympy#18482](#) Incorrect evaluation of limit
- [sympy/sympy#18499](#) The result of $(1/\infty)^{**}(-\infty)$ should be ∞
- [sympy/sympy#18501](#) Extraneous variable in limit result
- [sympy/sympy#18508](#) `NotImplementedError` in limit
- [sympy/sympy#18507](#) Bug in `Mul`
- [sympy/sympy#18707](#) There is a problem or limitation when the Limit is calculated
- [sympy/sympy#18751](#) handling of `rsolve` coefficients
- [sympy/sympy#18749](#) polys: Berlekamp factorization failure
- [sympy/sympy#18895](#) Factor with `extension=True` drops a factor of $y - 1$
- [sympy/sympy#18894](#) `sring extension=True` error: nan is not in any domain
- [sympy/sympy#18531](#) `apart`: hangs or takes too long
- [sympy/sympy#14806](#) `Domain.is_positive` (and friends) is a wrong interface
- [sympy/sympy#18874](#) Zero divisor from `sring` over $\mathbb{Q}[\sqrt{2} + \sqrt{5}]$
- [sympy/sympy#16620](#) Slow `factor(x^n-1, modulus=2)` computation for some “difficult” n
- [sympy/sympy#18997](#) Incorrect limit result involving `Abs`, returns expression involving a symbol
- [sympy/sympy#18992](#) Possibly incorrect limit related to Stirling’s formula
- [sympy/sympy#19026](#) Bug in `Limit`
- [sympy/sympy#12303](#) Ellipse comparison with other geometric entities throws an error
- [sympy/sympy#11986](#) Typo Error in `mathml.py`
- [sympy/sympy#12361](#) Misspelling of “Bareiss” in `Matrix` module
- [sympy/sympy#12452](#) `is_upper()` raises `IndexError` for tall matrices
- [sympy/sympy#19070](#) bug in `poly`
- [sympy/sympy#16971](#) `is_extended_real` should not evaluate if sign is not known

8.6 Diofant 0.10

27 Jan 2019

8.6.1 New features

- New representation for elements of *AlgebraicField* (page 429), see [#619](#), [#631](#) and [#763](#).
- Support towers of algebraic field extensions: ground domain for *AlgebraicField* (page 429) can be also an instance of *AlgebraicField* (page 429), see [#653](#).
- New subclasses of *AlgebraicField* (page 429): *RealAlgebraicField* (page 429) and *ComplexAlgebraicField* (page 429), see [#669](#), [#630](#) and [#748](#). Thanks to Kalevi Suominen for help with review.

- Added `integer_digits()` (page 92), see #765.
- `FiniteField` (page 429) support prime power orders, forbid everything else, see #622 and #762.

8.6.2 Major changes

- Stable enumeration of polynomial roots in `RootOf` (page 540), see #633, #658, #741 and #768. Thanks to Kalevi Suominen for the implementation idea and help with review.
- Support root isolation for polynomials with algebraic coefficients, see #673 and #630. Thanks to Kalevi Suominen for help with review.
- Polynomials with algebraic coefficients will use algebraic number domains per default, see #478.

8.6.3 Compatibility breaks

- Removed DMF class, see #620.
- Removed `K[x, y, ...]` sugar, use `poly_ring()` (page 427) to create polynomial rings, see #622.
- Removed `FracField` class, see #622.
- `get_field()` method for domains, derived from `CommutativeRing` (page 428), now is a property, e.g. `field` (page 427), see #622.
- Removed `PolyRing` class, see #621.
- `get_ring()` method for domains, derived from `CommutativeRing` (page 428), now is a property, e.g. `ring` (page 428), see #621.
- Removed `compose` option for `minimal_polynomial()` (page 539), use method instead, see #624.
- `field_isomorphism()` (page 539) take fields as arguments, see #627.
- Functions `minimal_polynomial()` (page 539) and `primitive_element()` (page 539) return `PurePoly` (page 527) instances, see #628.
- Removed ANP class, see #619.
- Removed `to_number_field()`, use `convert()` (page 427) instead, see #619.
- Removed `RealNumber` alias, see #635.
- Method `characteristic()` now is a property of `CharacteristicZero` (page 429) and `FiniteField` (page 429), see #636.
- Removed `of_type()`, `abs()`, `is_one()`, `unify_with_symbols()` and `map()` methods and `has_CharacteristicZero` attribute of `Domain` (page 427), see #636, #704 and #637.
- Removed `is_unit()`, `numer()` and `denom()` methods of `CommutativeRing` (page 428), see #637.
- `from_<Foo>()` methods of `Domain` (page 427) now are private, see #637.
- Method `from_expr()` (page 427) was renamed from `from_diofant()`, see #637.
- Method `to_expr()` (page 427) was renamed from `to_diofant()`, see #637.

- Removed AlgebraicNumber class, see #631.
- Removed polys.distributedmodules module, see #648.
- Removed p and q properties of *Rational* (page 88), see #654.
- Removed @public decorator, see #666.
- Removed dummy_eq() method from *Basic* (page 46), see #666.
- *Subs* (page 128) now support only Subs(expr, (var1, val1), (var2, val2), ...) syntax, see #667.
- *RootOf* (page 540) don't canonicalize anymore polynomials to have integer coefficients, use *expand_func()* (page 135) instead, see #679.
- Removed Theano support, see #681.
- Removed minpoly alias for *minimal_polynomial()* (page 539), see #684.
- Method *set_order()* (page 506) was renamed from fglm(), see #688.
- Removed row(), col(), row_del() and col_del() methods of *Matrix* (page 432), see #688.
- Removed add() and mul() methods for *PolynomialRing* (page 429), see #697.
- Removed itercoeffs(), itermonoms(), iterterms(), listcoeffs(), listmonoms(), listterms(), const(), imul_num() and square() methods of *PolyElement* (page 758), see #697.
- Removed abs(), neg(), add(), add_ground(), sub(), sub_ground(), mul(), mul_ground(), pow(), sqr(), nth(), factor_list_include(), revert(), gff(), gff_list(), sqf_list_include(), homogenize(), homogeneous_order(), eq() and ne() methods of *Poly* (page 507), see #688, #701, #732, #717, #727, #729 and #747.
- *subs()* (page 52) support one argument (a mapping or an iterable of pairs), see #532.
- Renamed is_sqf property of *Poly* (page 507) to *is_squarefree* (page 519), see #724.
- Removed all option for *sqf_list()* (page 524) method, see #727.
- Renamed has_Ring/Field attributes of *Domain* (page 427) to is_Ring/Field, see #729.
- Removed symmetric option for polynomial functions, see #761.
- Removed print_mathml() function and tree submodule, see #769.
- Removed zero option from *as_dict()* (page 509) method, see #771.
- Removed lift() method of *Poly* (page 507), see #771.

8.6.4 Minor changes

- Be sure that *minimal_polynomial()* (page 539) returns an irreducible polynomial over specified domain, see #622.
- Support algebraic function fields in *minpoly_groebner()* (page 771), see #623.
- Added argument method for *minimal_polynomial()* (page 539) and MINPOLY_METHOD configuration option to select default algorithm, see #624.
- Support derivatives of *RootOf* (page 540) instances, see #624.

- `primitive_element()` (page 539) now return an algebraic integer and support algebraic fields, see #643, #655 and #659.
- Support `conjugate` (page 278), `Abs` (page 276), `re` (page 275) and `im` (page 275) in `minimal_polynomial()` (page 539), see #661 and #668.
- `refine()` (page 541) method to refine interval for the root, see #670.
- Support detection of imaginary roots in `RootOf` (page 540), see #625.
- Mutable matrices support indexed deletion with `__delitem__()`, see #688.
- Integer powers of `RootOf` (page 540) instances are automatically reduced, according to their minimal polynomial, see #691.
- Support gmpy2.mpz ground type for numerator/denominator of `Rational` (page 88), see #694.
- Added FALLBACK_GCD_ZZ_METHOD configuration option to specify GCD algorithm for polynomials with integer coefficients if heuristic GCD was off or just unlucky, see #721.
- Added GCD_AA_METHOD configuration option to specify GCD algorithm for polynomials with algebraic coefficients, see #721.
- `sqf_part()` (page 524), `sqf_norm()` (page 524), `sqf_list()` (page 524) methods and `is_squarefree` (page 519) property use notion of being square-free w.r.t. to all polynomial variables, see #726.
- 100% test coverage for `core` (page 41), `polys` (page 505) and stats modules. Overall test coverage is around 97%.

8.6.5 Developer changes

- Removed cachetools dependence, see #647.
- Depend on `pylint`, see #668.
- Use `setuptools_scm` to track package versions, see #725.
- Don't use doctests for code coverage statistics, see #739.

8.6.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These Sympy issues also were addressed:

- [sympy/sympy#14384](#) An unspecified power of x is reported to be $O(\log(x)^{**6})$
- [sympy/sympy#14393](#) Incorrect limit
- [sympy/sympy#14414](#) Should `QQ[x, y, ...]` syntax be removed?
- [sympy/sympy#13886](#) Raise an exception for non-prime p in `FiniteField(p)`
- [sympy/sympy#14220](#) Should be there both `PolyRing` and `PolynomialRing`?
- [sympy/sympy#7724](#) roots should find the roots of $x^{**4}I + x^{**2} + I$
- [sympy/sympy#5850](#) `minpoly()` should use `PurePoly`
- [sympy/sympy#14494](#) make better decisions for `minpoly` based on domain

- [sympy/sympy#14389](#) AlgebraicNumber should be a domain element?
- [sympy/sympy#14291](#) `poly(((x - 1)**2 + 1)*((x - 1)**2 + 2)*(x - 1)).all_roots()` hangs
- [sympy/sympy#14590](#) `limit((n**3*((n + 1)/n)**n)/((n + 1)*(n + 2)*(n + 3)), n, oo)` is incorrect
- [sympy/sympy#14645](#) Bug when solving multivariate polynomial systems with identical equations
- [sympy/sympy#14294](#) `to_number_field` should be idempotent for single extension
- [sympy/sympy#14721](#) `solve` can't find solution
- [sympy/sympy#14293](#) Sorting of polynomial roots
- [sympy/sympy#14380](#) `AlgebraicField.numer()` could return an algebraic integer
- [sympy/sympy#14442](#) Should `AlgebraicField` be a Composite domain?
- [sympy/sympy#14759](#) `dup_isolate_real_roots_list()` docstring is wrong
- [sympy/sympy#14738](#) `dup_count_complex_roots()` can't handle degenerate cases
- [sympy/sympy#14782](#) `integrate(sqrt(-x**2 + 1)*(-x**2 + x), [x, -1, 1])` is incorrect
- [sympy/sympy#14791](#) No solution is returned for `solve(exp(log(5)*x) - exp(log(2)*x), x)`
- [sympy/sympy#14793](#) Limit involving `log(factorial(x))` incorrect
- [sympy/sympy#14811](#) Exception during evaluation of limit (only locally, not in the live version)
- [sympy/sympy#14822](#) `RisingFactorial` cannot do numerical (floating point) evaluations
- [sympy/sympy#14820](#) octave/matlab codegen wrong for two argument `zeta`
- [sympy/sympy#14831](#) `minpoly(-3*sqrt(12*sqrt(2) + 17) + 12*sqrt(2) + 17 - 2*sqrt(2)*sqrt(12*sqrt(2) + 17), x)` fails
- [sympy/sympy#14476](#) `QQ.algebraic_field(Rational)` should be just `QQ`
- [sympy/sympy#14885](#) Sympy series gives `TypeError` on `x^(-3/2) * exp(x)` at `x = 0`
- [sympy/sympy#15055](#) Incorrect limit of $n^3 * ((-n - 1) * \sin(1/n) + (n + 2) * \sin(1/(n + 1))) / (-n + 1)$
- [sympy/sympy#15056](#) `dsolve: get_numbered_constants` should consider Functions
- [sympy/sympy#6938](#) Undefined Functions should not use the `evalf` name lookup scheme
- [sympy/sympy#8945](#) `integrate(sin(x)**3/x, (x, 0, 1))` can't do it
- [sympy/sympy#15146](#) Incorrect limit $(n/2) * (-2*n^3 - 2*(n^3 - 1) * n^2 * \text{digamma}(n^3 + 1) + 2*(n^3 - 1) * n^2 * \text{digamma}(n^3 + n + 1) + n + 3)$
- [sympy/sympy#5934](#) `PolynomialError` with `minpoly()`
- [sympy/sympy#8210](#) Zero degree polynomial `copy()` error
- [sympy/sympy#11775](#) `TypeError: unorderable types: PolyElement() < mpz() from factor_list`
- [sympy/sympy#7047](#) Python and gmpy ground type specific stuff from "from sympy import *"
- [sympy/sympy#15323](#) limit of the derivative of $(1-1/x)^x$ as $x \rightarrow 1+$ gives wrong answer
- [sympy/sympy#15344](#) `mathematica_code` gives wrong output with `Max`

- [sympy/sympy#12602](#) `count_roots` is extremely slow with Python ground types
- [sympy/sympy#5595](#) Should `mpmath` use the polys ground types?
- [sympy/sympy#5602](#) Poly should use `free_symbols` to check for variable dependence
- [sympy/sympy#5555](#) Explain coefficient domain handling in `groebner()`'s docstring
- [sympy/sympy#15407](#) BUG: `dsolve` fails for linear first order ODE with three equations
- [sympy/sympy#15311](#) 3rd-order ODE with irrational coefficient fails
- [sympy/sympy#11668](#) Get rid of bare "except"s
- [sympy/sympy#4511](#) `integrate(cos(x)**2 / (1-sin(x)))` gives too complicated answer
- [sympy/sympy#15474](#) `dsolve` system gives complicated solution for diagonal system
- [sympy/sympy#15502](#) Python 3.7 test failures
- [sympy/sympy#15520](#) 5th-order ODE with irrational coefficient fails
- [sympy/sympy#15539](#) Order at negative infinity
- [sympy/sympy#15561](#) SymPy's `Number.__divmod__` doesn't agree with the builtin `divmod`
- [sympy/sympy#15574](#) `dsolve` fails for a system of independent equations
- [sympy/sympy#12695](#) [matrices] remove dead files `densearith.py` `densetools.py` and `densesolve.py`
- [sympy/sympy#5428](#) Should Poly use an algebraic domain by default?
- [sympy/sympy#14337](#) Poly constructor uses domain EX when it's not necessary
- [sympy/sympy#8818](#) `lambdify` precision loss with `module=mpmath` from high-precision Floats
- [sympy/sympy#9544](#) Finite fields
- [sympy/sympy#15798](#) `Poly.copy()` does not copy unused generators
- [sympy/sympy#15810](#) `integrate(1/(2**(2*x/3)+1), (x,0,oo))` is wrong

8.7 Diofant 0.9

23 Feb 2018

8.7.1 New features

- Polynomial solvers now express all available solutions with *RootOf* (page 540), see [#400](#).
- Added *mod_inverse()* (page 91) and *invert()* (page 69), see [#390](#). Thanks to Chris Smith.
- Support solving linear programming problems, see [#283](#) and [#461](#).
- Added *interval* (page 541) property for *RootOf* (page 540), see [#508](#).
- Added AST transformation *IntegerDivisionWrapper* (page 566) to wrap integer division, see [#519](#).
- Added AST transformation *FloatRationalizer* (page 566) to wrap *float*'s, see [#538](#).

- Compute `independent_sets` (page 505) and dimension of the ideal, generated by Gröbner basis, see #573.
- Added `permutedims()` (page 700) and `derive_by_array()` (page 700), see #567. Thanks to Francesco Bonazzi.
- Added `is_square()` (page 250), `ordered_partitions()` (page 738), `permute_signs()` (page 741) and `signed_permutations()` (page 743), see #578. Thanks to Chris Smith.

8.7.2 Major changes

- Assumptions (old) moved from `Basic` (page 46) to `Expr` (page 57), see #311.
- `solve()` (page 606) now return `list` of `dict`'s, see #473.
- `diofant.polys.domains` module is now top-level module `domains` (page 426), see #487.
- Optionally reduce `RootOf` (page 540) instances to have polynomials with integer coefficients, see #430.
- `solve_poly_system()` (page 609) now able to solve positive-dimensional systems, see #448 and #573.
- Big update of the `diophantine` (page 610) module with a lot of bugfixes, see #578. Thanks to Chris Smith.

8.7.3 Compatibility breaks

- Removed `assumption0` property, see #382.
- `check_assumptions()` (page 44) moved to `assumptions` (page 44), see #387.
- Removed `nsolve()` function, see #387.
- `is_comparable` (page 70) and `is_hypergeometric()` (page 72) moved to `Expr` (page 57), see #391.
- Removed `solve_triangulated()` and `solve_biquadratic()` functions, `solve_poly_system()` (page 609) now use `dict` as output, see #389 and #448.
- Removed support for solving undetermined coefficients in `solve()` (page 606), see #389.
- Removed `intersect()` alias for `intersection()` (page 568), see #396.
- Removed `interactive_traversal()`, see #395.
- Removed `xring()` and `xfield()`, see #403.
- Removed `jcode` submodule and `TableForm` class, see #403.
- Removed `agca` submodule of `polys` (page 505), see #404.
- Removed `pager_print()` and `print_fcode()`, see #411.
- Disallow “increase” precision of `Float` (page 85)’s with `evalf()` (page 138), see #380.
- Removed `experimental_lambdify()` and `intervalmath` module from plotting package, see #384.
- Removed `solve()` (page 606) flags `set`, `manual`, `minimal`, `implicit`, `particular`, `quick`, `exclude`, `force` and `numerical` see #426, #554 and #549.

- Removed support for inequalities in `solve()` (page 606), please use `reduce_inequalities()` (page 610) instead, see #426.
- Removed `get_domain()` method of `Poly` (page 507), use `domain` (page 512) property instead, see #479.
- Renamed `prec` argument of `Float` (page 85) to `dps`, see #510.
- Removed `as_content_primitive()` method of `Basic` (page 46), see #529.
- Removed `canonical_variables()` property to `canonical_variables()` (page 65), see #534.
- Removed `group` option of `find()` (page 48), which now return a `dict`, see #529.
- Removed support for Python 3.4, see #543.
- Second argument of `checksol()` (page 695) must be a `dict`. See #549.
- Removed `solve_undetermined_coeffs()` function, see #554.
- Make `matches()` method for `Basic` (page 46) - private, see #557.
- Removed `replace()` (page 49) flags `simultaneous` and `map`, see #557.
- Make `strict=True` - default for `evalf()` (page 138), see #537.
- Removed `I` property of the `MatrixExpr` (page 500), see #577.
- Removed `isolate()` function, see #585.
- `gcd()` (page 531) and `lcm()` (page 533) now are two-arg functions, see #585.
- Removed `is_zero_dimensional()` function and `GroebnerBasis` (page 505)'s property of the same name, use `dimension` (page 505) instead, see #573.
- Removed `MonomialOps` class, see #586.
- Renamed `n` argument of `evalf()` (page 138) to `dps`, see #596.
- Return representation of elements via primitive in `primitive_element()` (page 539) (former `ex=True` format), see #597.
- Removed `pprint_try_use_unicode()` function, see #605.

8.7.4 Minor changes

- New integration heuristics for integrals with `Abs` (page 276), see #321.
- Support unevaluated `RootOf` (page 540), see #400.
- Sorting of symbolic quadratic roots now same as in `RootOf` (page 540) for numerical coefficients, see #400.
- Improve printing of Mathematica code, see #400, #433, #438, #519, #553 and #571.
- Support simple first-order DAE with `dsolve()` (page 635) helper `ode_lie_group()` (page 661), see #413.
- Added support for limits of relational expressions, see #414.
- Make `MatrixSymbol` (page 501) truly atomic, see #415.
- Support rewriting `Min` (page 299) and `Max` (page 299) as `Piecewise` (page 297), see #426.

- `minimal_polynomial()` (page 539) fixed to support generic AlgebraicNumber's, see #433 and #438.
- AlgebraicNumber now support arithmetic operations, see #428 and #485.
- Support rewrite `RootOf` (page 540) via radicals, see #563.
- Export set singletons, see #577.
- Correct implementation of the trial method (uses Gröbner bases) in `primitive_element()` (page 539), see #608 and #609.
- Support (not in `RootOf` (page 540) yet) of root isolation for polynomials over Gaussian rationals, see #606.
- 100% test coverage for `matrices` (page 432), `domains` (page 426), `logic` (page 417), `parsing` (page 748) and `printing` (page 548) modules. Overall test coverage is 96%.

8.7.5 Developer changes

- Enabled docstring testing with flake8, see #408.
- Use only relative imports in the codebase, see #421.
- Enabled flake8-comprehensions plugin, see #420.
- Imports are sorted with isort, see #520.
- Depend on hypothesis, see #547.
- Depend on pytest-xdist, see #551.
- Depend on pytest-timeout, see #608.

8.7.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release. These Sympy issues also were addressed:

- [sympy/sympy#11879](#) Strange output from common limit used in elementary calculus
- [sympy/sympy#11884](#) Addition with Order gives wrong result
- [sympy/sympy#11045](#) `integrate(1/(x*sqrt(x**2-1)), (x, 1, 2))` Sympy latest version `AttributeError: 'Or' object has no attribute 'Its'`
- [sympy/sympy#7165](#) `integrate(abs(y - x**2), (y,0,2))` raises `ValueError: gamma function pole`
- [sympy/sympy#8733](#) `integrate(abs(x+1), (x, 0, 1))` raises gamma function pole error
- [sympy/sympy#8430](#) `integrate(abs(x), (x, 0, 1))` does not simplify
- [sympy/sympy#12005](#) `Subs._eval_derivative` doubles derivatives
- [sympy/sympy#11799](#) Something wrong with the Riemann tensor?
- [sympy/sympy#12018](#) solution not found by `Sum` and `gospersum`
- [sympy/sympy#5649](#) Bug with `AlgebraicNumber.__eq__`
- [sympy/sympy#11538](#) Bug in `solve` maybe

- [sympy/sympy#12081](#) `integrate(x**(-S(3)/2)*exp(-x), (x, 0, oo))` encounters Runtime Error
- [sympy/sympy#7214](#) Move old assumptions from Basic to Expr
- [sympy/sympy#4678](#) Have `solve()` return `RootOf` when it can't solve equations
- [sympy/sympy#7789](#) `Poly(...).all_roots` fails for general quadratic equation
- [sympy/sympy#8255](#) `roots_quadratic` should return roots in same order as `Poly.all_roots(radicals=False)`
- [sympy/sympy#7138](#) How to solve system of differential equations with symbolic solution?
- [sympy/sympy#11691](#) Test failing with matplotlib 2.0.0
- [sympy/sympy#7457](#) `TypeError` when using both multiprocessing and gmpy
- [sympy/sympy#12115](#) Cannot access imported submodules in `sympy.core`
- [sympy/sympy#4315](#) series expansion of piecewise fails
- [sympy/sympy#6807](#) `atoms` does not work correctly in the otherwise case of `Piecewise`
- [sympy/sympy#12114](#) `solve()` leads to `ZeroDivisionError`: polynomial division
- [sympy/sympy#5169](#) All elements of `.args` should be Basic
- [sympy/sympy#6249](#) Problems with `MatrixSymbol` and simplifying functions
- [sympy/sympy#6426](#) `test_args.py` should also test rebuildability
- [sympy/sympy#11461](#) `NameError`: name 'Ne' is not defined plotting `real_root((log(x/(x-2))), 3)`
- [sympy/sympy#10925](#) plot doesn't work with `Piecewise`
- [sympy/sympy#12180](#) Confusing output from `sympy.solve`
- [sympy/sympy#5786](#) `factor(extension=[I])` gives wrong results
- [sympy/sympy#9607](#) `factor` - incorrect result
- [sympy/sympy#8754](#) Problem factoring trivial polynomial
- [sympy/sympy#8697](#) `rsolve` fails to find solutions to some higher order recurrence relations
- [sympy/sympy#8694](#) Match fail
- [sympy/sympy#8710](#) `geometry's` `encloses` method fails for non-polygons
- [sympy/sympy#10337](#) bad Boolean args not rejected
- [sympy/sympy#9447](#) `sets.Complement` fails on certain Unions
- [sympy/sympy#10305](#) Complement Of Universal Subsets
- [sympy/sympy#10413](#) `ascii pprint` of `ProductSet` uses non-ascii multiplication symbol
- [sympy/sympy#10414](#) `pprint(Union, use_unicode=False)` raises error (but `str(Union)` works)
- [sympy/sympy#10375](#) `lambdify` on `sympy.Min` does not work with NumPy
- [sympy/sympy#10433](#) `Dict` does not accept `collections.defaultdict`
- [sympy/sympy#9044](#) pretty printing: Trace could be improved (and LaTeX)
- [sympy/sympy#10445](#) Improper integral does not evaluate

- [sympy/sympy#10379](#) dsolve() converts floats to integers/rationals
- [sympy/sympy#10633](#) Eq(True, False) doesn't evaluate
- [sympy/sympy#7163](#) integrate((sign(x - 1) - sign(x - 2))*cos(x), x) raises TypeError: doit() got an unexpected keyword argument 'manual'
- [sympy/sympy#11881](#) ZeroDivisionError: pole in hypergeometric series random test failure
- [sympy/sympy#11801](#) Exception when printing Symbol("")
- [sympy/sympy#11911](#) typo in docs of printing
- [sympy/sympy#10489](#) Mathematical Symbol does not seem to serialize correctly LaTeX printer
- [sympy/sympy#10336](#) nsimplify problems with oo and inf
- [sympy/sympy#12345](#) nonlinsolve (solve_biquadratic) gives no solution with radical
- [sympy/sympy#12375](#) sympy.series() is broken?
- [sympy/sympy#5514](#) Poly(x, x) * I != I * Poly(x, x)
- [sympy/sympy#12398](#) Limits With abs in certain cases remains unevaluated
- [sympy/sympy#12400](#) polytool.poly() can't raise polynomial to negative power?
- [sympy/sympy#12221](#) Issue with definite piecewise integration
- [sympy/sympy#12522](#) BooleanTrue and Boolean False should have simplify method
- [sympy/sympy#12555](#) limit((3**x + 2 * x**10) / (x**10 + E**x), x, -oo) gives 0 instead of 2
- [sympy/sympy#12569](#) problem with polygamma or im
- [sympy/sympy#12578](#) Taylor expansion wrong (likely because of wrong substitution at point of evaluation?)
- [sympy/sympy#12582](#) Can't solve integrate(abs(x**2-3*x), (x, -15, 15))
- [sympy/sympy#12747](#) Missing constant coefficient in Taylor series of degree 1
- [sympy/sympy#12769](#) Slow limit() calculation?!
- [sympy/sympy#12942](#) Remove x**1.0 == x hack from core
- [sympy/sympy#12238](#) match can take a long time (possibly forever)
- [sympy/sympy#4269](#) ordering of classes
- [sympy/sympy#13081](#) Some comparisons between rational and irrational numbers are incorrect
- [sympy/sympy#13078](#) Return NotImplemented, not False, upon rich comparison with unknown type
- [sympy/sympy#13098](#) sympy.floor() sometimes returns the wrong answer
- [sympy/sympy#13312](#) SymPy does not evaluate integrals of exponentials with symbolic parameter and limit
- [sympy/sympy#13111](#) Don't use "is" to compare classes
- [sympy/sympy#10488](#) integrate(x/(a*x+b), x) gives wrong answer
- [sympy/sympy#9706](#) Interval(-oo, 0).closure hangs

- [sympy/sympy#10740](#) Add a test for `Interval(..) in Interval(..) == False`
- [sympy/sympy#10592](#) `zeta(0, n)` where `n` is negative is wrong
- [sympy/sympy#7858](#) Nth root mod giving wrong solutions
- [sympy/sympy#5412](#) `N(oo*I)` returns wrong result
- [sympy/sympy#10710](#) Any dict-like object in `expr.subs`
- [sympy/sympy#10810](#) Implemented function gives `ValueError` when constructing float expression in `sympy 1.0`
- [sympy/sympy#10867](#) Getting `KeyError` while solving ode : `dsolve(Eq(g(x).diff(x).diff(x), (x-2)**2 +(x-3)**3), g(x))`
- [sympy/sympy#10782](#) `condition_number()` for empty matrices giving `ValueError`
- [sympy/sympy#10719](#) `eigenvals` of empty matrix raises `IndexError`
- [sympy/sympy#10680](#) unable to get unevaluated Integral object for `integrate (x**log(x**log(x**log(x))) , x)`
- [sympy/sympy#10701](#) Is the empty matrix nilpotent? `IndexError: Index out of range: a[0]`
- [sympy/sympy#10770](#) Adding a row or a column to an empty matrix
- [sympy/sympy#10773](#) `sympify` evaluates Div Operation in case of Unary Operator when `evaluate = False`
- [sympy/sympy#13332](#) `limit()`: `AttributeError: 'NoneType' object has no attribute 'rewrite'`
- [sympy/sympy#13382](#) Incorrect Result for `limit(n*(((n+1)**2+1)/(n**2+1)-1), n ,oo)`
- [sympy/sympy#13403](#) Incorrect Result for `limit(n*(-1 + (n + log(n + 1) + 1)/(n + log(n))), n ,oo)`
- [sympy/sympy#13416](#) Incorrect Result for `limit((-n**3*log(n)**3 + (n - 1)*(n + 1)**2*log(n + 1)**3)/(n**2*log(n)**3), n ,oo)`
- [sympy/sympy#13462](#) Bug in `sympy.limit()`
- [sympy/sympy#13501](#) Incorrect integral of a rational function with a symbolic coefficient
- [sympy/sympy#13536](#) `TypeError` for integration from infinity to a positive value
- [sympy/sympy#13545](#) Poly loses modulus after arithmetic
- [sympy/sympy#13460](#) Integration of certain cubic rational functions is incorrect
- [sympy/sympy#13071](#) `meijerg.is_number` is wrong
- [sympy/sympy#13575](#) `limit(acos(erfi(x)), x, 1)` causes recursion error
- [sympy/sympy#13629](#) bug in `rsolve`
- [sympy/sympy#13645](#) `sympy` hangs on evaluating expression
- [sympy/sympy#7067](#) `factor_list()` error Python3
- [sympy/sympy#11378](#) `S.Reals` should be accessible as just "Reals"
- [sympy/sympy#10999](#) `diop: holzer` error
- [sympy/sympy#11000](#) `diop: power_representation`
- [sympy/sympy#11026](#) `diophantine(x**3+y**3-2) -> KeyError` instead of `{(1, 1)}`

- [sympy/sympy#8943](#) diophantine misses trivial solution
- [sympy/sympy#11016](#) diop: sum of squares needs to try more options to satisfy conditions
- [sympy/sympy#9538](#) diophantine() doesn't let you specify the variable order
- [sympy/sympy#11049](#) diop: recursion error
- [sympy/sympy#11021](#) diop: power_representation(4**5, 3, 1) -> (4,)
- [sympy/sympy#11050](#) diop: partition(n, k) gives redundant result
- [sympy/sympy#13853](#) Why does the expansion of polylog(1, z) have exp_polar(-I*pi)?
- [sympy/sympy#13849](#) solve/nonlinsolve: RuntimeError: run out of coefficient configurations
- [sympy/sympy#9366](#) rootof: Constructing RootOfs with polys containing RootOf coefficients
- [sympy/sympy#13914](#) The power of zoo
- [sympy/sympy#14000](#) sqrt and other root functions should inherit from Function
- [sympy/sympy#11099](#) Min and Max would not substitute in evalf
- [sympy/sympy#8257](#) Interval(-oo, oo) + FiniteSet(oo) takes forever
- [sympy/sympy#11198](#) factor_list(sqrt(const)*x) error
- [sympy/sympy#10784](#) autowrap on windows - distutils doesn't work with C compiler
- [sympy/sympy#10897](#) rewrite im() in terms of re() and vice versa
- [sympy/sympy#10963](#) x**6000%400 hangs
- [sympy/sympy#10931](#) S.Integers - S.Integers does not evaluate
- [sympy/sympy#2799](#) S.UniversalSet + Interval(0, oo) takes forever
- [sympy/sympy#11090](#) ImmutableMatrix * MatrixSymbol raises AttributeError
- [sympy/sympy#11207](#) floor(ceiling(x)) doesn't simplify
- [sympy/sympy#9135](#) Incorrect substitution of partial derivatives by .subs()
- [sympy/sympy#10829](#) subs method gives wrong result for powers
- [sympy/sympy#10816](#) is_nthpow_residue(a,n,m) gives NotImplemented error when m don't have primitive root
- [sympy/sympy#10886](#) No solution by nthroot_mod
- [sympy/sympy#10157](#) Replace needs_brackets with parenthesize in the latex printer
- [sympy/sympy#10972](#) [tensor module] incorrect evaluation of TensMul.data
- [sympy/sympy#10044](#) Error pretty printing a tuple with a sympy.vector basis vector
- [sympy/sympy#10395](#) nfloat changes the arguments inside Max.
- [sympy/sympy#10641](#) Or, And don't evaluate
- [sympy/sympy#10821](#) latex bug for commutator output
- [sympy/sympy#9296](#) simplify(a)+simplify(b) Is Not simplify(a+b)
- [sympy/sympy#9630](#) simplify() rounds a numerical coefficient (indeed very close to unity) to 1

- [sympy/sympy#12792](#) Simplify with float values leads to non-equal result
- [sympy/sympy#12506](#) Simplify() returns wrong simplified expressions using Sympy 1.0 (trigonometric functions)
- [sympy/sympy#13115](#) Bug in simplify ?
- [sympy/sympy#13149](#) factor() of expression with float coefficients gives incorrect result
- [sympy/sympy#14117](#) Run out of coefficient configurations in primitive_element()
- [sympy/sympy#14159](#) Can't set bottom and top bounds of root isolation rectangle with dup_isolate_complex_roots_sqf()
- [sympy/sympy#11122](#) $x > 0$ doesn't evaluate for $x = \text{Symbol}('x', \text{positive}=\text{False})$
- [sympy/sympy#11418](#) diophantine: misclassification
- [sympy/sympy#9862](#) [tensor] error when retrieving data from TensAdd instance involving fully contracted tensor and scalar
- [sympy/sympy#11525](#) [tensor] TensAdd ignores all but one scalar argument
- [sympy/sympy#11530](#) ITE(x, True, False) should auto simplify to x
- [sympy/sympy#11559](#) str of Transpose should be valid Python
- [sympy/sympy#11547](#) mathml(Matrix([0,1,2])) gives back error
- [sympy/sympy#11306](#) numpy lambdify of piecewise doesn't work for invalid values
- [sympy/sympy#7171](#) sin(x).rewrite(pow) raises RuntimeError: maximum recursion depth
- [sympy/sympy#2866](#) lambdify inserts numpy after math
- [sympy/sympy#11351](#) TypeError exception in totient and reduced_totient LaTeX printers
- [sympy/sympy#14289](#) Sign of generator of an algebraic numberfield

8.8 Diofant 0.8

7 Nov 2016

8.8.1 New features

- MrvAsympt algorithm to find asymptotic expansion, see [aseries\(\)](#) (page 64) method and [#6](#). Thanks to Avichal Dayal.
- [findrecur\(\)](#) (page 264) method to find recurrence relations (with Sister Celine's algorithm), see [#15](#).
- Support for [Pow](#) (page 99)/[log](#) (page 296) branch-cuts in limits, see [#140](#).
- Added basic optimization package, see [minimize\(\)](#) (page 753) and [#108](#).
- Cartesian product of iterables using Cantor pairing, see [cantor_product\(\)](#) (page 731) and [#276](#).
- [Rationals](#) (page 579) set, [#255](#).

- New simple and robust solver for systems of linear ODEs, see [#286](#). Thanks to Colin B. Macdonald.
- Added mutable/immutable N-dim arrays, sparse and dense, see [#275](#).
- `dsolve()` (page 635) now support initial conditions for ODEs, see [#307](#). Thanks to Aaron Meurer.

8.8.2 Major changes

- Depend on `setuptools`, see [#44](#).
- *The Gruntz Algorithm* (page 787) reimplemented correctly, see [#68](#).
- Replaced `exp(x)` with `E**x` internally, see [#79](#).
- Used `srepr()` (page 561) instead of `sstr()` (page 561) for `__repr__()` printing, see [#39](#).
- Major cleanup for series methods, see [#187](#).
- Depend on `cachetools` to implement caching, see [#72](#) and [#209](#).
- Assumption system (old) was validated ([#316](#) and [#334](#)) and improved:
 - 0 now is imaginary, see [#8](#)
 - `extended_real` fact added, reals are finite now, see [#36](#)
 - complex are finite now, see [#42](#).
 - added docstrings for assumption properties, see [#354](#).

8.8.3 Compatibility breaks

- Removed physics submodule, see [#23](#).
- Removed galgebra submodule, see [#45](#).
- Removed pyglet plotting, see [#50](#).
- Removed TextBackend from plotting, see [#67](#).
- Removed SageMath support, see [#84](#).
- Removed unify submodule, see [#88](#).
- Removed crypto submodule, see [#102](#).
- Removed `print_gtk`, see [#114](#).
- Unbundle strategies module, see [#103](#).
- Removed “old” argument for `match/matches`, see [#141](#).
- Removed `when_multiple` kwarg in `Piecewise`, see [#156](#).
- Support for Python 2 was removed, see [#160](#).
- Removed `core.py`, see [#60](#) and [#164](#).
- Removed `S(foo)` syntax, see [#115](#).
- Removed (new) assumptions submodule, see [#122](#).
- Removed undocumented `Symbol.__call__`, see [#201](#).

- Removed categories and liealgebras submodules, see [#280](#).
- Rename module sympy -> diofant, see [#315](#).
- Use gmpy2, drop gmpy support, see [#292](#).
- Removed redundant dom properties in polys, see [#308](#).
- Removed manualintegrate function, see [#279](#).

8.8.4 Minor changes

- Add support for bidirectional limits, see [#10](#).
- Reimplement `cot` (page 283), see [#113](#).
- A better implementation of `singularities()` (page 752), see [#147](#).
- Fix “flip” of arguments in relational expressions, see [#30](#).
- Make Gosper code use new dispersion algorithm, see [#205](#). Thanks to Raoul Bourquin.
- Consolidate code for solving linear systems, see [#253](#).
- Hacks for automatic symbols and wrapping int’s replaced with AST transformers, see [#278](#) and [#167](#).
- Build correct inhomogeneous solution in `rsolve_hyper()` (page 685), see [#298](#).
- Evaluate matrix powers for non-diagonalizable matrices, see [#275](#).
- Support non-orthogonal Jordan blocks, see [#275](#).
- Make `risch_integrate(x**x, x)` work, see [#275](#).
- Support CPython 3.6, see [#337](#) and [#356](#).

8.8.5 Developer changes

- Unbundle numpydoc, see [#26](#).
- Deprecate AUTHORS file, all credits go to the aboutus.rst, see [#87](#).
- Use python’s `tokenize()`, see [#120](#).
- Drop using bundled pytest fork, depend on pytest for testing, see [#38](#), [#152](#), [#91](#), [#48](#), [#90](#), [#96](#) and [#99](#).
- Adopt No Code Of Conduct, see [#212](#).
- Measure code coverage, enable codecov.io reports. See [#217](#).
- Adopt pep8 ([#2](#)) and then flake8 ([#214](#)) for code quality testing.
- Add regression tests with `DIOFANT_USE_CACHE=False` [#323](#).
- Add interface tests, see [#219](#) and [#307](#).
- Test for no DeprecationWarning in the codebase, see [#356](#).

8.8.6 Issues closed

See the [release milestone](#) for complete list of issues and pull requests involved in this release.

These Sympy issues also were addressed:

- [sympy/sympy#9351](#) order-1 series wrong with non-zero expansion point
- [sympy/sympy#9034](#) Unicode printing problem with mixture of logs and powers
- [sympy/sympy#7927](#) pretty print incorrect result with powers of sin
- [sympy/sympy#9283](#) KroneckerDelta(p, 0) raises IndexError
- [sympy/sympy#9274](#) Wrong Jordan form: complex eigenvalues w/ geo. mult. > alg. mult.
- [sympy/sympy#9398](#) Simplify of small imaginary number yields 0
- [sympy/sympy#7259](#) LambertW has no series expansion at x=0 (nan)
- [sympy/sympy#9832](#) $x^2 < \infty$ returns True but $x < \infty$ un-evaluated for real x
- [sympy/sympy#9053](#) MatMul(2, Matrix(...)).doit() doesn't do it
- [sympy/sympy#9052](#) $\text{trace}(2A) \neq 2\text{Trace}(A)$ because LHS still has an MatMul
- [sympy/sympy#9533](#) Logical operators in octave_code
- [sympy/sympy#9545](#) Mod(zoo, 0) causes RunTime Error
- [sympy/sympy#9652](#) Fail in plot_implicit test on OSX 10.8.5
- [sympy/sympy#8432](#) Tests fail, seems like Cython is not configured to compile with numpy correctly
- [sympy/sympy#9542](#) codegen octave global vars should print "global foo" at top of function
- [sympy/sympy#9326](#) Bug with Dummy
- [sympy/sympy#9413](#) Circularity in assumptions of products
- [sympy/sympy#8840](#) sympy fails to construct $(1 + x)x$ with disabled cache
- [sympy/sympy#4898](#) Replace $\exp(x)$ with E^x internally
- [sympy/sympy#10195](#) Simplification bug on alternating series.
- [sympy/sympy#10196](#) reduce_inequalities error
- [sympy/sympy#10198](#) solving abs with negative powers
- [sympy/sympy#7917](#) Implement cot as a ReciprocalTrigonometricFunction
- [sympy/sympy#8649](#) If t is transcendental, t^n is determined (wrongly) to be non-integer
- [sympy/sympy#5641](#) Compatibility with py.test
- [sympy/sympy#10258](#) Relational involving Piecewise evaluates incorrectly as True
- [sympy/sympy#10268](#) solving inequality involving exp fails for large values
- [sympy/sympy#10237](#) improper inequality reduction
- [sympy/sympy#10255](#) solving a Relational involving Piecewise fails
- [sympy/sympy#10290](#) Computing series where the free variable is not just a symbol is broken

- [sympy/sympy#10304](#) Equality involving expression with known real part and 0 should evaluate
- [sympy/sympy#9471](#) Wrong limit with log and constant in exponent
- [sympy/sympy#9449](#) limit fails with “maximum recursion depth exceeded” / Python crash
- [sympy/sympy#8462](#) Trivial bounds on binomial coefficients
- [sympy/sympy#9917](#) $O(n \sin(n) + 1, (n, \infty))$ returns $O(n \sin(n), (n, \infty))$
- [sympy/sympy#7383](#) Integration error
- [sympy/sympy#7098](#) Incorrect expression resulting from integral evaluation
- [sympy/sympy#10323](#) bad ceiling(sqrt(big integer))
- [sympy/sympy#10326](#) Interval(-oo, oo) contains oo
- [sympy/sympy#10095](#) simplify((1/(2*E))**oo) returns nan
- [sympy/sympy#4187](#) integrate(log(x)*exp(x), (x, 0, oo)) should return -EulerGamma
- [sympy/sympy#10383](#) det of empty matrix is 1
- [sympy/sympy#10382](#) limit(fibonacci(n + 1)/fibonacci(n), n, oo) does not give GoldenRatio
- [sympy/sympy#10388](#) factorial2 runs into RuntimeError for non-integer
- [sympy/sympy#10391](#) solve((2*x + 8)*exp(-6*x), x) can't find any solution
- [sympy/sympy#8241](#) Wrong assumption/result in a parametric limit
- [sympy/sympy#3539](#) Symbol.__call__ should not create a Function
- [sympy/sympy#7216](#) Limits involving branch cuts of elementary functions not handled
- [sympy/sympy#10503](#) Series return an incorrect result
- [sympy/sympy#10567](#) Integral(v,t).doit() differs from integrate(v,t)
- [sympy/sympy#9075](#) sympy.limit yields incorrect result
- [sympy/sympy#10610](#) limit(3**n*3**(-n - 1)*(n + 1)**2/n**2, n, oo) is wrong
- [sympy/sympy#4173](#) implement maximize([x**(1/x), x>0], x)
- [sympy/sympy#10803](#) Bad pretty printing of power of Limit
- [sympy/sympy#10836](#) Latex generation error for .series expansion for rightharpoon term
- [sympy/sympy#9558](#) Bug with limit
- [sympy/sympy#4949](#) solve_linear_system contains duplicate rref algorithm
- [sympy/sympy#5952](#) Standard sets (ZZ, QQ, RR, etc.) for the sets module
- [sympy/sympy#9608](#) Partition can't be ordered
- [sympy/sympy#10961](#) fractional order Laguerre gives wrong result
- [sympy/sympy#10976](#) incorrect answer for limit involving erf
- [sympy/sympy#10995](#) acot(-x) evaluation
- [sympy/sympy#11011](#) Scientific notation should be delimited for LaTeX
- [sympy/sympy#11062](#) Error while simplifying equations containing csc and sec using trigsimp_groebner

- [sympy/sympy#10804](#) $1/\text{limit}(\text{airybi}(x)*\text{root}(x, 4)*\exp(-2*x**(S(3)/2)/3), x, \infty)**2$ is wrong
- [sympy/sympy#11063](#) Some wrong answers from `rsolve`
- [sympy/sympy#9480](#) `Matrix.rank()` incorrect results
- [sympy/sympy#10497](#) `next(iter(S.Integers*S.Integers))` hangs (expected (0, 0), ...)
- [sympy/sympy#5383](#) Calculate limit error
- [sympy/sympy#11270](#) Limit erroneously reported as infinity
- [sympy/sympy#5172](#) `limit()` throws `TypeError: an integer is required`
- [sympy/sympy#7055](#) Failures in `rsolve_hyper` from `test_rsolve_bulk()`
- [sympy/sympy#11261](#) Recursion solver fails
- [sympy/sympy#11313](#) Series of Derivative
- [sympy/sympy#11290](#) `1st_exact_Integral` wrong result
- [sympy/sympy#10761](#) $(1/(x**-2 + x**-3)).\text{series}(x, 0)$ gives wrong result
- [sympy/sympy#10024](#) `Eq(Mod(x, 2*pi), 0)` evaluates to `False`
- [sympy/sympy#7985](#) Indexed should work with subs on a container
- [sympy/sympy#9637](#) `S.Reals - FiniteSet(n)` returns `EmptySet - FiniteSet(n)`
- [sympy/sympy#10003](#) $P(X < -1)$ of `ExponentialDistribution`
- [sympy/sympy#10052](#) $P(X < \infty)$ for any Continuous Distribution raises `AttributeError`
- [sympy/sympy#10063](#) Integer raised to Float power does not evaluate
- [sympy/sympy#10075](#) `X.pdf(x)` for Symbol `x` returns 0
- [sympy/sympy#9823](#) Matrix power of identity matrix fails
- [sympy/sympy#10156](#) do not use `has()` to test against `self.variables` when factoring `Sum`
- [sympy/sympy#10113](#) `imageset(lambda x: x**2/(x**2 - 4), S.Reals)` returns (1, ∞)
- [sympy/sympy#10020](#) $\infty**I$ raises `RunTimeError`
- [sympy/sympy#10240](#) `Not(And(x>2, x<3))` does not evaluate
- [sympy/sympy#8510](#) Differentiation of general functions
- [sympy/sympy#10220](#) `Matrix.jordan_cells()` fails
- [sympy/sympy#10092](#) subs into inequality involving `RootOf` raises `GeneratorsNeeded`
- [sympy/sympy#10161](#) `factor` gives an invalid expression
- [sympy/sympy#10243](#) Run the examples during automated testing or at release
- [sympy/sympy#10274](#) The helpers `kwarg` in `autowrap` method is probably broken.
- [sympy/sympy#10210](#) LaTeX printing of Cycle
- [sympy/sympy#9539](#) `diophantine(6*k + 9*n + 20*m - x)` gives `TypeError: unsupported operand type(s) for *: 'NoneType' and 'Symbol'`
- [sympy/sympy#11407](#) Series expansion of the square root gives wrong result
- [sympy/sympy#11413](#) Wrong result from Matrix norm
- [sympy/sympy#11434](#) Matrix `rank()` produces wrong result

- [sympy/sympy#11526](#) Different result of limit after simplify
- [sympy/sympy#11553](#) Polynomial solve with GoldenRatio causes Traceback
- [sympy/sympy#8045](#) make all NaN is_* properties that are now None -> False (including is_complex)
- [sympy/sympy#11602](#) Replace dots with ldots or cdots
- [sympy/sympy#4720](#) Initial conditions in dsolve()
- [sympy/sympy#11623](#) Wrong groebner basis
- [sympy/sympy#10292](#) poly cannot generically be rebuilt from its args
- [sympy/sympy#6572](#) Remove “#doctest: +SKIP” comments on valid docstrings
- [sympy/sympy#10134](#) Remove “raise StopIteration”
- [sympy/sympy#11672](#) limit(Rational(-1,2)**k, k, oo) fails
- [sympy/sympy#11678](#) Invalid limit of floating point matrix power
- [sympy/sympy#11746](#) undesired (wrong) substitution behavior in sympy?
- [sympy/sympy#3904](#) missing docstrings in core
- [sympy/sympy#3112](#) Asymptotic expansion
- [sympy/sympy#9173](#) Series/limit fails unless expression is simplified first.
- [sympy/sympy#9808](#) Complements with symbols should remain unevaluated
- [sympy/sympy#9341](#) Cancelling very long polynomial expression
- [sympy/sympy#9908](#) Sum(1/(n**3 - 1), (n, -oo, -2)).doit() raise UnboundLocalVariable
- [sympy/sympy#6171](#) Limit of a piecewise function
- [sympy/sympy#9276](#) ./bin/diagnose_imports: does it work at all?!
- [sympy/sympy#10201](#) Solution of “first order linear non-homogeneous ODE-System” is wrong
- [sympy/sympy#9057](#) segfault on printing Integral of phi(t)
- [sympy/sympy#11159](#) Substitution with E fails
- [sympy/sympy#2839](#) init_session(auto_symbols=True) and init_session(auto_int_to_Integer=True) do not work
- [sympy/sympy#11081](#) where possible, use python fractions for Rational
- [sympy/sympy#10974](#) solvers.py contains BOM character
- [sympy/sympy#10806](#) LaTeX printer: Integral not surrounded in brackets
- [sympy/sympy#10801](#) Make limit work with binomial
- [sympy/sympy#9549](#) series expansion: (x**2 + x + 1)/(x**3 + x**2) about oo gives wrong result
- [sympy/sympy#4231](#) add a test for complex integral from wikipedia
- [sympy/sympy#8634](#) limit(x**n, x, -oo) is sometimes wrong
- [sympy/sympy#8481](#) Wrong error raised trying to calculate limit of Poisson PMF
- [sympy/sympy#9956](#) Union(Interval(-oo, oo), FiniteSet(1)) not evaluated

- [sympy/sympy#9747](#) test_pieewise_lambdify fails locally
- [sympy/sympy#7853](#) Deprecation of lambdify converting Matrix -> numpy.matrix
- [sympy/sympy#9634](#) Repeated example in the docstring of hermite
- [sympy/sympy#8500](#) Using and operator vs fuzzy_and while querying assumptions
- [sympy/sympy#9192](#) $O(y + 1) = O(1)$
- [sympy/sympy#7130](#) Definite integral returns an answer with indefinite integrals
- [sympy/sympy#8514](#) Inverse Laplace transform of a simple function fails after updating from 0.7.5 to 0.7.6
- [sympy/sympy#9334](#) Numexpr must be string argument to lambdify
- [sympy/sympy#8229](#) `limit((x**Rational(1,4)-2)/(sqrt(x)-4)**Rational(2, 3), x, 16)` `NotImplementedError`
- [sympy/sympy#8061](#) `limit(4**(acos(1/(1+x**2))**2)/log(1+x, 4), x, 0)` raises `NotImplementedError`
- [sympy/sympy#7872](#) Substitution in Order fails
- [sympy/sympy#3496](#) limits for complex variables
- [sympy/sympy#2929](#) `limit((x*exp(x))/(exp(x)-1), x, -oo)` gives `-oo`
- [sympy/sympy#8203](#) Why is `oo` real?
- [sympy/sympy#7649](#) `S.Zero.is_imaginary` should be `True`?
- [sympy/sympy#7256](#) use old assumptions in code
- [sympy/sympy#6783](#) Get rid of confusing assumptions
- [sympy/sympy#5662](#) `AssocOp._eval_template_is_attr` is wrong or misused
- [sympy/sympy#5295](#) Document assumptions
- [sympy/sympy#4856](#) coding style
- [sympy/sympy#4555](#) use pyflakes to identify simple bugs in sympy and fix them
- [sympy/sympy#5773](#) Remove the `cmp_to_key()` helper function
- [sympy/sympy#5484](#) use `sort_key` instead of old comparison system
- [sympy/sympy#8825](#) Can't use both `weakref's` & `cache`
- [sympy/sympy#8635](#) `limit(x**n-x**(n-k), x, oo)` sometimes raises `NotImplementedError`
- [sympy/sympy#8157](#) Non-informative error raised when computing limit of `cos(n*pi)`
- [sympy/sympy#7599](#) Addition of expression and order term fails
- [sympy/sympy#6179](#) wrong order in series
- [sympy/sympy#5415](#) limit involving multi-arg function (polygamma) fails
- [sympy/sympy#2865](#) `gruntz` doesn't work properly for big-O with `point!=0`
- [sympy/sympy#5907](#) `integrate(1/(x**2 + a**2)**2, x)` is wrong if `a` is real
- [sympy/sympy#11722](#) `series()` calculation up to $O(t^k)$ returns invalid coefficients for $t^k \cdot \log(t)$
- [sympy/sympy#8804](#) series expansion of $1/x$ ignores order parameter

- [sympy/sympy#10728](#) `Dummy(commutative=False).is_zero` -> False

8.9 SymPy releases

For convenience, here included release notes from the old SymPy versions, up to 0.7.6. Changelog sources are taken from [web archive](#) and adapted to resemble usual Diofant's release notes.

8.9.1 SymPy 0.7.6

21 Nov 2014

New features

- New module `calculus.finite_diff` for generating finite difference formulae approximating derivatives of arbitrary order on arbitrarily spaced grids.
- New module `physics.optics` for symbolic computations related to optics.
- `geometry` module now supports 3D geometry.
- Support for series expansions at a point other than 0 or ∞ . See [sympy/sympy#2427](#).
- Rules for the intersection of integer ImageSets were added. See [sympy/sympy#7587](#). We can now do things like $\{2 \cdot m \mid m \in \mathbb{Z}\} \cap \{3 \cdot n \mid n \in \mathbb{Z}\} = \{6 \cdot t \mid t \in \mathbb{Z}\}$ and $\{2 \cdot m \mid m \in \mathbb{Z}\} \cap \{2 \cdot n + 1 \mid n \in \mathbb{Z}\} = \emptyset$.
- `dsolve` module now supports system of ODEs including linear system of ODEs of 1st order for 2 and 3 equations and of 2nd order for 2 equations. It also supports homogeneous linear system of n equations.
- New support for continued fractions, including iterators for partial quotients and convergents, and reducing a continued fraction to a Rational or a quadratic irrational.
- Support for Egyptian fraction expansions, using several different algorithms.
- Addition of generalized linearization methods to `physics.mechanics`.
- Use an LRU cache by default instead of an unbounded one. See [sympy/sympy#7464](#). Control cache size by the environment variable `SYMPY_CACHE_SIZE` (default is 500). `SYMPY_CACHE_SIZE=None` restores the unbounded cache.
- Added `fastcache` as an optional dependency. Requires v0.4 or newer. Control via `SYMPY_CACHE_SIZE`. May result in significant speedup. See [sympy/sympy#7737](#).
- New experimental module `physics.unitsystems` for computation with dimensions, units and quantities gathered into systems. This opens the way to dimensional analysis and better quantity calculus. The old module `physics.units` will stay available until the new one reaches a mature state. See [sympy/sympy#2628](#).
- New `Complement` class to represent relative complements of two sets. See [sympy/sympy#7462](#).
- New trigonometric functions (`asec`, `acsc`), many enhancements for other trigonometric functions (see [sympy/sympy#7500](#)).

- New `Contains` class to represent the relation “is an element of” (see [sympy/sympy#7989](#)).
- The code generation tools (code printers, codegen, autowrap, and ufuncify) have been updated to support a wider variety of constructs, and do so in a more robust way. Major changes include added support for matrices as inputs/outputs, and more robust handling of conditional (Piecewise) statements.
- `ufuncify` now uses a backend that generates actual `numpy.ufuncs` by default through the use of the `numpy C api`. This allows broadcasting on *all* arguments. The previous `cython` and `f2py` backends are still accessible through the use of the `backend` kwarg.
- `CodeGen` now generates code for Octave and Matlab from SymPy expressions. This is supported by a new `CodePrinter` with interface `octave_code`. For example `octave_code(Matrix([[x**2, sin(pi*x*y), ceiling(x)]]))` gives the string `[x.^2 sin(pi*x.*y) ceil(x)]`.
- New general 3D vector package at `sympy.vector`. This package provides a 3D vector object with the `Del`, `gradient`, `divergence`, `curl`, and operators. It supports arbitrary rotations of Cartesian coordinate systems and arbitrary locations of points.

Compatibility breaks

- All usage of inequalities (`>`, `>=`, `<`, `<=`) on SymPy objects will now return SymPy’s `S.true` or `S.false` singletons instead of Python’s `True` or `False` singletons. Code that checks for, e.g., `(a < b) is True` should be changed to `(a < b) == True` or `(a < b) == S.true`. Use of `is` is not recommended here.
- The `subset()` method in `sympy.core.sets` is marked as being deprecated and will be removed in a future release ([sympy/sympy#7460](#)). Instead, the `is_subset()` method should be used.
- Previously, if you compute the series expansion at a point other than 0, the result was shifted to 0. Now SymPy returns the usual series expansion, see [sympy/sympy#2427](#).
- In `physics.mechanics`, `KanesMethod.linearize` has a new interface. Old code should be changed to use this instead. See docstring for information.
- `physics.gaussopt` has been moved to `physics.optics.gaussopt`. You can still import it from the previous location but it may result in a deprecation warning.
- This is the last release with the bundled [mpmath library](#). In the next release you will have to install this library from the official site.
- Previously `lambdify` would convert `Matrix` to `numpy.matrix` by default. This behavior is being deprecated, and will be completely phased out with the release of 0.7.7. To use the new behavior now set the `modules` kwarg to `[{'ImmutableMatrix': numpy.array}, 'numpy']`. If `lambdify` will be used frequently it is recommended to wrap it with a partial as so: `lambdify = functools.partial(lambdify, modules=[{'ImmutableMatrix': numpy.array}, 'numpy'])`. For more information see [sympy/sympy#7853](#) and the `lambdify` docstring.
- `Set.complement` doesn’t exist as an attribute anymore. Now we have a method `Set.complement(<universal_set>)` which complements the given universal set.
- Removed `is_finite` assumption (see [sympy/sympy#7891](#)). Use instead a combination of `is_bounded` and `is_nonzero` assumptions.

- `is_bounded` and `is_unbounded` assumptions were renamed to `is_finite` and `is_infinite` (see [sympy/sympy#7947](#)).
- Removed `is_infinitesimal` assumption (see [sympy/sympy#7995](#)).
- Removed `is_real` property for Sets, use `Set.is_subset(Reals)` instead (see [sympy/sympy#7996](#)).
- For generic symbol `x` (SymPy's symbols are not bounded by default), inequalities with `oo` are no longer evaluated as they were before, e.g. `x < oo` no longer evaluates to `True`). See [sympy/sympy#7861](#).
- `CodeGen` has been refactored to make it easier to add other languages. The main high-level tool is still `utilities.codegen.codegen`. But if you previously used the `Routine` class directly, note its `__init__` behaviour has changed; the new `utilities.codegen.make_routine` is recommended instead and by default retains the previous C/Fortran behaviour. If needed, you can still instantiate `Routine` directly; it only does minimal sanity checking on its inputs. See [sympy/sympy#8082](#).
- `FiniteSet([1, 2, 3, 4])` syntax not supported anymore, use `FiniteSet(1, 2, 3, 4)` instead. See [sympy/sympy#7622](#).

Minor changes

- Updated the parsing module to allow sympification of lambda statements to their SymPy equivalent.
- `Lambdify` can now use `numexpr` by specifying `modules='numexpr'`.
- Use with `evaluate(False)` context manager to control automatic evaluation. E.g. with `evaluate(False): x + x` is actually `x + x`, not `2*x`.
- `IndexedBase` and `Indexed` are changed to be commutative by default.
- `sympy.core.sets` moved to `sympy.sets`.
- Changes in `sympy.sets`:
 - `Infinite Range` is now allowed. See [sympy/sympy#7741](#).
 - `is_subset()`: The `is_subset()` method deprecates the `subset()` method. `self.is_subset(other)` checks if `self` is a subset of `other`. This is different from `self.subset(other)`, which checked if `other` is a subset of `self`.
 - `is_superset()`: A new method `is_superset()` method is now available. `self.is_superset(other)` checks if `self` is a superset of `other`.
 - `is_proper_subset` and `is_proper_superset`: Two new methods allow checking if one set is the proper subset and proper superset of another respectively. For e.g. `self.is_proper_subset(other)` and `self.is_proper_superset(other)` checks if `self` is the proper subset of `other` and if `self` is the proper superset of `other` respectively.
 - `is_disjoint()`: A new method for checking if two sets are disjoint.
 - `powerset()`: A new method `powerset()` has been added to find the power set of a set.
 - The cardinality of a `ProductSet` can be found using the `len()` function.
- Changes in `sympy.plot.plot_implicit`:

- The `plot_implicit` function now also allows explicitly specifying the symbols to plot on the X and Y axes. If not specified, the symbols will be assigned in the order they are sorted.
- The `plot_implicit` function also allows axes labels for the plot to be specified.
- rules for simplification of ImageSet were added [sympy/sympy#7625](#). As a result $\{x \mid x \in \mathbb{Z}\}$ now simplifies to \mathbb{Z} and $\{\sin(n) \mid n \in \{\tan(m) \mid m \in \mathbb{Z}\}\}$ automatically simplifies to $\{\sin(\tan(m)) \mid m \in \mathbb{Z}\}$.
- `coth(0)` now returns Complex Infinity. See [sympy/sympy#7634](#).
- `dioptr` is added to `physics.units`. See [sympy/sympy#7782](#).
- `replace` now respects commutativity, see [sympy/sympy#7752](#).
- The `CCodePrinter` gracefully handles Symbols which have string representations that match C reserved words, see [sympy/sympy#8199](#).
- `limit` function now returns an unevaluated `Limit` instance if it can't compute given limit, see [sympy/sympy#8213](#).

8.9.2 SymPy 0.7.5

22 Feb 2014

Major changes

- The version of `mpmath` included in SymPy has been updated to 0.18.
- New routines for efficiently compute the *dispersion* of a polynomial or a pair thereof.
- Fancy indexing of matrices is now provided, e.g. `A[:, [1, 2, 5]]` selects all rows and only 3 columns.
- Enumeration of multiset partitions is now based on an implementation of Algorithm 7.1.2.5M from Knuth's *The Art of Computer Programming*. The new version is much faster, and includes fast methods for enumerating only those partitions with a restricted range of sizes, and counting multiset partitions. (See the new file `sympy.utilities.enumerative.py`.)
- `distance` methods were added to `Line` and `Ray` to compute the shortest distance to them from a point.
- The `normal_lines` method was added to `Ellipse` to compute the lines from a point that strike the `Ellipse` at a normal angle.
- `inv_quick` and `det_quick` were added as functions in `solvers.py` to facilitate fast solution of small symbolic matrices; their use in `solve` has reduced greatly the time needed to solve such systems.
- `solve_univariate_inequality` has been added to `sympy.solvers.inequalities.py`.
- `as_set` attribute for `Relationals` and `Booleans` has been added.
- Several classes and functions strictly associated with vector calculus were moved from `sympy.physics.mechanics` to a new package `sympy.physics.vector`. (See [sympy/sympy#2732](#), [sympy/sympy#2862](#) and [sympy/sympy#2894](#)).

- New implementation of the Airy functions Ai and Bi and their derivatives Ai' and Bi' (called `airyai`, `airybi`, `airyaiprime` and `airybiprime`, respectively). Most of the usual features of SymPy special function are present. Notable exceptions are Gruntz limit computation helpers and meijerg special functions integration code.
- Euler-Lagrange equations (function `euler_equations`) in a new package `sympy.calculus` ([sympy/sympy#2431](#)).

Compatibility breaks

- the `submatrix` method of matrices was removed; access the functionality by providing slices or list of rows/columns to matrix directly, e.g. `A[:, [1, 2]]`.
- `Matrix([])` and `Matrix([[]])` now both return a 0x0 matrix
- `terms_gcd` no longer removes a -1.0 from expressions
- `extract_multiplicatively` will not remove a negative Number from a positive one, so `(4*x*y).extract_multiplicatively(-2*x)` will return `None`.
- the shape of the result from `M.cross(B)` now has the same shape as matrix `M`.
- The factorial of negative numbers is now `zoo` instead of `0`. This is consistent with the definition $\text{factorial}(n) = \text{gamma}(n + 1)$.
- `1/0` returns `zoo`, not `oo` ([sympy/sympy#2813](#)).
- `zoo.is_number` is `True` ([sympy/sympy#2823](#)).
- `oo < I` raises `TypeError`, just as for finite numbers ([sympy/sympy#2734](#)).
- `1**oo == nan` instead of `1`, better documentation for `Pow` class ([sympy/sympy#2606](#)).

Minor changes

- Some improvements to the gamma function.
- `generate_bell` now generates correct permutations for any number of elements.
- It is no longer necessary to provide `nargs` to objects subclassed from `Function` unless an `eval` class method is not defined. (If `eval` is defined, the number of arguments will be inferred from its signature.)
- geometric Point creation will be faster since simplification is done only on Floats
- Some improvements to the intersection method of the Ellipse.
- solutions from solve of equations involving multiple log terms are more robust
- `idiff` can now return higher order derivatives
- Added `to_matrix()` method to `sympy.physics.vector.Vector` and `sympy.physics.dyadic.Dyadic`. ([sympy/sympy#2686](#)).
- Printing improvements for `sympy.physics.vector` objects and mechanics printing. (See [sympy/sympy#2687](#), [sympy/sympy#2728](#), [sympy/sympy#2772](#), [sympy/sympy#2862](#) and [sympy/sympy#2894](#)).
- Functions with LaTeX symbols now print correct LaTeX. ([sympy/sympy#2772](#)).

- `init_printing` has several new options, including a flag `print_builtint` to prevent SymPy printing of basic Python types ([sympy/sympy#2683](#)), and flags to let you supply custom printers ([sympy/sympy#2894](#)).
- improvements in evaluation of `imageset` for Intervals ([sympy/sympy#2723](#)).
- Set properties to determine boundary and interior ([sympy/sympy#2744](#)).
- input to a function created by `lambdify` no longer needs to be flattened.

8.9.3 SymPy 0.7.4

9 Dec 2013

Major changes

- Python 3
 - SymPy now uses a single code-base for Python 2 and Python 3.
- Geometric Algebra
 - The internal representation of a multivector has been changes to more fully use the inherent capabilities of SymPy. A multivector is now represented by a linear combination of real commutative SymPy expressions and a collection of non-commutative SymPy symbols. Each non-commutative symbol represents a base in the geometric algebra of an N-dimensional vector space. The total number of non-commutative bases is $2^N - 1$ (N of which are a basis for the vector space) which when including scalars give a dimension for the geometric algebra of 2^N . The different products of geometric algebra are implemented as functions that take pairs of bases symbols and return a multivector for each pair of bases.
 - The LaTeX printing module for multivectors has been rewritten to simply extend the existing sympy LaTeX printing module and the sympy LaTeX module is now used to print the bases coefficients in the multivector representation instead of writing an entire LaTeX printing module from scratch.
 - The main change in the geometric algebra module from the viewpoint of the user is the interface for the gradient operator and the implementation of vector manifolds:
 - * The gradient operator is now implemented as a special vector (the user can name it `grad` if they wish) so the if `F` is a multivector field all the operations of `grad` on `F` can be written `grad*F`, `F*grad`, `grad^F`, `F^grad`, `grad|F`, `F|grad`, `grad<F`, `F<grad`, `grad>F`, and `F>grad` where `*`, `^`, `|`, `<`, and `>` are the geometric product, outer product, inner product, left contraction, and right contraction, respectively.
 - * The vector manifold is defined as a parametric vector field in an embedding vector space. For example a surface in a 3-dimensional space would be a vector field as a function of two parameters. Then multivector fields can be defined on the manifold. The operations available to be performed on these fields are directional derivative, gradient, and projection. The weak point of the current manifold representation is that all fields on the manifold are represented in terms of the bases of the embedding vector space.
- Classical Cryptography, implements:
 - Affine ciphers

- Vigenere ciphers
- Bifid ciphers
- Hill ciphers
- RSA and “kid RSA”
- linear feedback shift registers.
- Common Subexpression Elimination (CSE). Major changes have been done in cse internals resulting in a big speedup for larger expressions. Some changes reflect on the user side:
 - Adds and Muls are now recursively matched ($[w*x, w*x*y, w*x*y*z]$ now turns into $[(x0, w*x), (x1, x0*y)], [x0, x1, x1*z]$)
 - CSE is now not performed on the non-commutative parts of multiplications (it avoids some bugs).
 - Pre and post optimizations are not performed by default anymore. The `optimizations` parameter still exists and `optimizations='basic'` can be used to apply previous default optimizations. These optimizations could really slow down cse on larger expressions and are no guarantee of better results.
 - An `order` parameter has been introduced to control whether Adds and Muls terms are ordered independently of hashing implementation. The default `order='canonical'` will independently order the terms. `order='none'` will not do any ordering (hashes order is used) and will represent a major performance improvement for really huge expressions.
 - In general, the output of cse will be slightly different from the previous implementation.
- Diophantine Equation Module. This is a new addition to SymPy as a result of a GSOC project. With the current release, following five types of equations are supported.
 - Linear Diophantine equation, $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$
 - General binary quadratic equation, $ax^2 + bxy + cy^2 + dx + ey + f = 0$
 - Homogeneous ternary quadratic equation, $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$
 - Extended Pythagorean equation, $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
 - General sum of squares, $x_1^2 + x_2^2 + \dots + x_n^2 = k$
- Unification of Sum, Product, and Integral classes
 - A new superclass has been introduced to unify the treatments of indexed expressions, such as Sum, Product, and Integral. This enforced common behavior across the objects, and provides more robust support for a number of operations. For example, Sums and Integrals can now be factored or expanded. `S.subs()` can be used to substitute for expressions inside a Sum/Integral/Product that are independent of the index variables, including unknown functions, for instance, `Integral(f(x), (x, 1, 3)).subs(f(x), x**2)`, while `Sum.change_index()` or `Integral.transform` are now used for other changes of summation or integration variables. Support for finite and infinite sequence products has also been restored.
 - In addition there were a number of fixes to the evaluation of nested sums and sums involving Kronecker delta functions, see [sympy/sympy#7023](#) and [sympy/sympy#7086](#).
- Series

- The `Order` object used to represent the growth of a function in series expansions as a variable tend to zero can now also represent growth as a variable tend to infinity. This also fixed a number of issues with limits. See [sympy/sympy#3333](#) and [sympy/sympy#5769](#).
- Division by `Order` is disallowed, see [sympy/sympy#4855](#).
- Addition of `Order` object is now commutative, see [sympy/sympy#4279](#).
- Physics
 - Initial work on gamma matrices, depending on the tensor module.
- Logic
 - New objects `true` and `false` which are Basic versions of the Python builtins `True` and `False`.
- Other
 - Arbitrary comparisons between expressions (like $x < y$) no longer have a boolean truth value. This means code like `if x < y` or `sorted(exprs)` will raise `TypeError` if $x < y$ is symbolic. A typical fix of the former is `if (x < y) is True` (assuming the `if` block should be skipped if $x < y$ is symbolic), and of the latter is `sorted(exprs, key=default_sort_key)`, which will order the expressions in an arbitrary, but consistent way, even across platforms and Python versions. See [sympy/sympy#5931](#).
 - Arbitrary comparisons between complex *numbers* (for example, $I > 1$) now raise `TypeError` as well (see [sympy/sympy#2510](#)).
 - `minimal_polynomial` now works with algebraic functions, like `minimal_polynomial(sqrt(x) + sqrt(x + 1), y)`.
 - `exp` can now act on any matrix, even those which are not diagonalizable. It is also more comfortable to call it, `exp(m)` instead of just `m.exp()`, as was required previously.
 - `sympify` now has an option `evaluate=False` that will not automatically simplify expressions like $x+x$.
 - Deep processing of `cancel` and `simplify` functions. `simplify` is now recursive through the expression tree. See e.g. [sympy/sympy#7022](#).
 - Improved the modularity of the codebase for potential subclasses, see [sympy/sympy#6751](#).
 - The SymPy cheatsheet was cleaned up.

Compatibility breaks

- Removed deprecated `Real` class and `is_Real` property of `Basic`, see [sympy/sympy#4820](#).
- Removed deprecated `'each_char'` option for `symbols()`, see [sympy/sympy#5018](#).
- The `viewer="StringIO"` option to `preview()` has been deprecated. Use `viewer="BytesIO"` instead. See [sympy/sympy#7083](#).
- `TransformationSet` has been renamed to `ImageSet`. Added public facing `imageset` function.

8.9.4 SymPy 0.7.3

13 Jul 2013

Major changes

- Integration
 - This release includes Risch integration algorithm from [Aaron Meurer's 2010 Google Summer of Code project](#). This makes `integrate` much more powerful and much faster for the supported functions. The algorithm is called automatically from `integrate()`. For now, only transcendental elementary functions containing `exp` or `log` are supported. To access the algorithm directly, use `integrate(expr, x, risch=True)`. The algorithm has the ability to prove that integrals are nonelementary. To determine if a function is nonelementary, integrate using `risch=True`. If the resulting `Integral` class is an instance of `NonElementaryIntegral`, then it is not elementary (otherwise, that part of the algorithm has just not been implemented yet).
- ODE
 - Built basic infrastructure of the PDE module ([sympy/sympy#1970](#))
- Theano Interaction
 - SymPy expressions can now be translated into [Theano](#) expressions for numeric evaluation. This includes most standard scalar operations (e.g. `sin`, `exp`, `gamma`, but not `beta` or `MeijerG`) and matrices. This system generally outperforms `lambdify` and `autowrap` but does require Theano to be installed.
- Matrix Expressions
 - Matrix expressions now support inference using the new assumptions system. New predicates include `invertible`, `symmetric`, `positive_definite`, `orthogonal`,
 - New operators include `Adjoint`, `HadamardProduct`, `Determinant`, `MatrixSlice`, `DFT`. Also, preliminary support exists for factorizations like `SVD` and `LU`.
- Context manager for New Assumptions
 - Added the `with assuming(*facts)` context manager for new assumptions. See [blogpost](#).

Compatibility breaks

- This is the last version of SymPy to support Python 2.5.
- The IPython extension, i.e., `%load_ext sympy.interactive.ipythonprinting` is deprecated. Use `from sympy import init_printing; init_printing()` instead. See [sympy/sympy#7013](#).
- The `viewer='file'` option to preview without a file name is deprecated. Use `file_name='name'` in addition to `viewer='file'`. See [sympy/sympy#7018](#).
- The deprecated syntax `Symbol('x', dummy=True)`, which had been deprecated since 0.7.0, has been removed. Use `Dummy('x')` or `symbols('x', cls=Dummy)` instead. See [sympy/sympy#6477](#).

- The deprecated Expr methods `as_coeff_terms` and `as_coeff_factors`, which have been deprecated in favor of `as_coeff_mul` and `as_coeff_add`, respectively (see also `as_coeff_Mul` and `as_coeff_Add`), were removed. The methods had been deprecated since SymPy 0.7.0. See [sympy/sympy#6476](#).
- The spherical harmonics have been completely rewritten. See [sympy/sympy#1510](#).

Minor changes

- Solvers
 - Added enhancements and improved the methods of solving exact differential equation. See [sympy/sympy#1955](#) and [sympy/sympy#1823](#).
 - Support for differential equations with linear coefficients and those that can be reduced to separable and linear form. See [sympy/sympy#1940](#), [sympy/sympy#1864](#) and [sympy/sympy#1883](#).
 - Support for first order linear general PDE's with constant coefficients ([sympy/sympy#2109](#)).
 - Return all found independent solutions for underdetermined systems.
 - Handle recursive problems for which $y(0) = 0$.
 - Handle matrix equations.
- Integration
 - `integrate` will split out integrals into Piecewise expressions when conditions must hold for the answer to be true. For example, `integrate(x**n, x)` now gives `Piecewise((log(x), Eq(n, -1)), (x**(n + 1)/(n + 1), True))` (previously it just gave `x**(n + 1)/(n + 1)`).
 - Calculate Gauss-Legendre and Gauss-Laguerre points and weights ([sympy/sympy#1497](#)).
 - Various new error and inverse error functions ([sympy/sympy#1703](#)).
 - Use `heurisch` for more symmetric and nicer results.
 - Gruntz for `expintegrals` and all new `erf*`.
 - `Li`, `li` logarithmic integrals ([sympy/sympy#1708](#)).
 - Integration of `li/Li` by `heurisch` ([sympy/sympy#1712](#)).
 - elliptic integrals, complete and incomplete.
 - Integration of complete elliptic integrals by `meijerg`.
 - Integration of Piecewise with symbolic conditions.
 - Fixed many wrong results of DiracDelta integrals.
- Logic
 - Addition of `SOPform` and `POSform` functions to `sympy.logic` to generate boolean expressions from truth tables.
 - Addition of `simplify_logic` function and enabling `simplify()` to reduce logic expressions to their simplest forms.

- Addition of `bool_equals` function to check equality of boolean expressions and return a mapping of variables from one `expr` to other that leads to the equality.
- Addition of disjunctive normal form methods - `to_dnf`, `is_dnf`
- Others
 - gmpy version 2 is now supported
 - Added `is_algebraic_expr()` method ([sympy/sympy#2176](#)).
 - Many improvements to the handling of noncommutative symbols:
 - * Better support in simplification functions, e.g. `factor`, `trigsimp`
 - * Better integration with `Order()`
 - * Better pattern matching
 - Improved pattern matching including matching the identity.
 - normalizes Jacobi polynomials
 - Quadrature rules for orthogonal polynomials in arbitrary precision (`hermite`, `laguerre`, `legendre`, `gen_legendre`, `jacobi`)
 - summation of harmonic numbers
 - Many improvements of the polygamma functions
 - evaluation at special arguments
 - Connections to harmonic numbers
 - structured full partial fraction decomposition (mainly interesting for developers)
 - `besselsimp` improvements
 - Karr summation convention
 - New spherical harmonics
 - improved `minimal_polynomial` using composition of algebraic numbers ([sympy/sympy#2038](#)).
 - faster integer polynomial factorization ([sympy/sympy#2148](#)).
 - Euler-Descartes method for quartic equations ([sympy/sympy#1947](#))
 - algebraic operations on tensors ([sympy/sympy#1700](#)).
 - tensor canonicalization ([sympy/sympy#1644](#)).
 - Handle the simplification of summations and products over a `KroneckerDelta`.
 - Implemented LaTeX printing of `DiracDelta`, `Heaviside`, `KroneckerDelta` and `LeviCivita`, also many `Matrix` expressions.
 - Improved LaTeX printing of fractions, `Mul` in general.
 - IPython integration and printing issues have been ironed out.
 - Stats now supports discrete distributions (e.g. `Poisson`) by relying on `Summation` objects
 - Added DOT printing for visualization of expression trees
 - Added information about solvability and nilpotency of named groups.

8.9.5 SymPy 0.7.2

16 Oct 2012

Major Changes

- Python 3 support
 - SymPy now supports Python 3. The officially supported versions are 3.2 and 3.3, but 3.1 should also work in a pinch. The Python 3-compatible tarballs will be provided separately, but it is also possible to download Python 2 code and convert it manually, via the bin/use2to3 utility. See the README for more.
- PyPy support
 - All SymPy tests pass in recent nightlies of PyPy, and so it should have full support as of the next version after 1.9.
- Combinatorics
 - A new module called Combinatorics was added which is the result of a successful GSoC project. It attempts to replicate the functionality of Combinatorica and currently has full featured support for Permutations, Subsets, Gray codes and Prufer codes.
 - In another GSoC project, facilities from computational group theory were added to the combinatorics module, mainly following the book “Handbook of computational group theory”. Currently only permutation groups are supported. The main functionalities are: basic properties (orbits, stabilizers, random elements...), the Schreier-Sims algorithm (three implementations, in increasing speed: with Jerrum’s filter, incremental, and randomized (Monte Carlo)), backtrack searching for subgroups with certain properties.
- Definite Integration
 - A new module called meijerint was added, which is also the result of a successful GSoC project. It implements a heuristic algorithm for (mainly) definite integration, similar to the one used in Mathematica. The code is automatically called by the standard integrate() function. This new algorithm allows computation of important integral transforms in many interesting cases, so helper functions for Laplace, Fourier and Mellin transforms were added as well.
- Random Variables
 - A new module called stats was added. This introduces a RandomSymbol type which can be used to model uncertainty in expressions.
- Matrix Expressions
 - A new matrix submodule named expressions was added. This introduces a MatrixSymbol type which can be used to describe a matrix without explicitly stating its entries. A new family of expression types were also added: Transpose, Inverse, Trace, and BlockMatrix. ImmutableMatrix was added so that explicitly defined matrices could interact with other SymPy expressions.
- Sets

- A number of new sets were added including atomic sets like `FiniteSet`, `Reals`, `Naturals`, `Integers`, `UniversalSet` as well as compound sets like `ProductSet` and `TransformationSet`. Using these building blocks it is possible to build up a great variety of interesting sets.
- Classical Mechanics
 - A physics submodule named `mechanics` was added which assists in formation of equations of motion for constrained multi-body systems. It is the result of 3 GSoC projects. Some nontrivial systems can be solved, and examples are provided.
- Quantum Mechanics
 - Density operator module has been added. The operator can be initialized with generic Kets or Qubits. The Density operator can also work with `TensorProducts` as arguments. Global methods are also added that compute entropy and fidelity of states. Trace and partial-trace operations can also be performed on these density operators.
 - To enable partial trace operations a `Tr` module has been added to the core library. While the functionality should remain same, this module is likely to be relocated to an alternate folder in the future. One can currently also use `sympy.core.Tr` to work on general trace operations, but this module is what is needed to work on trace and partial-trace operations on any `sympy.physics.quantum` objects.
 - The Density operators, `Tr` and Partial trace functionality was implemented as part of student participation in GSoC 2012.
 - Expanded angular momentum to include coupled-basis states and product-basis states. Operators can also be treated as acting on the coupled basis (default behavior) or on one component of the tensor product states. The methods for coupling and uncoupling these states can work on an arbitrary number of states. Representing, rewriting and applying states and operators between bases has been improved.
- Commutative Algebra
 - A new module `agca` was started which seeks to support computations in commutative algebra (and eventually algebraic geometry) in the style of `Macaulay2` and `Singular`. Currently there is support for computing Gröbner bases of modules over a (generalized) polynomial ring over a field. Based on this, there are algorithms for various standard problems in commutative algebra, e.g., computing intersections of submodules, equality tests in quotient rings, etc...
- Plotting Module
 - A new plotting module has been added which uses `Matplotlib` as its back-end. The plotting module has functions to plot the following:
 - * 2D line plots
 - * 2D parametric plots.
 - * 2D implicit and region plots.
 - * 3D surface plots.
 - * 3D parametric surface plots.
 - * 3D parametric line plots.
- Differential Geometry

- Thanks to a GSoC project the beginning of a new module covering the theory of differential geometry was started. It can be imported with `sympy.diffgeom`. It is based on “Functional Differential Geometry” by Sussman and Wisdom. Currently implemented are scalar, vector and form fields over manifolds as well as covariant and other derivatives.

Compatibility breaks

- The `KroneckerDelta` class was moved from `sympy/physics/quantum/kronecker.py` to `sympy/functions/special/tensor_functions.py`.
- Merged the `KroneckerDelta` class in `sympy/physics/secondquant.py` with the class above.
- The `Dij` class in `sympy/functions/special/tensor_functions.py` was replaced with `KroneckerDelta`.
- The errors raised for invalid float calls on SymPy objects were changed in order to emulate more closely the errors raised by the standard library. The `__float__` and `__complex__` methods of `Expr` are concerned with that change.
- The `solve()` function returns empty lists instead of `None` objects if no solutions were found. Idiomatic code of the form `sol = solve(...); if sol:...` will not be affected by this change.
- `Piecewise` no longer accepts a `Set` or `Interval` as a condition. One should explicitly specify a variable using `Set().contains(x)` to obtain a valid conditional.
- The statistics module has been deprecated in favor of the new stats module.
- `sympy/galgebra/GA.py`:
 - `set_main()` is no longer needed
 - `make_symbols()` is deprecated (use `sympy.symbols()` instead)
 - the symbols used in this package are no longer broadcast to the main program
- The classes for `Infinity`, `NegativeInfinity`, and `NaN` no longer subclass from `Rational`. Creating a `Rational` with 0 in the denominator will still return one of these classes, however.

Minor changes

- A new module `gaussopt` was added supporting the most basic constructions from Gaussian optics (ray tracing matrices, geometric rays and Gaussian beams).
- New classes were added to represent the following special functions: classical and generalized exponential integrals (`Ei`, `expint`), trigonometric (`Si`, `Ci`) and hyperbolic integrals (`Shi`, `Chi`), the polylogarithm (`polylog`) and the Lerch transcendent (`lerchphi`). In addition to providing all the standard sympy functionality (differentiation, numerical evaluation, rewriting ...), they are supported by both the new `meijerint` module and the existing hypergeometric function simplification module.
- An `ImmutableMatrix` class was created. It has the same interface and functionality of the old `Matrix` but is immutable and inherits from `Basic`.

- A new function in `geometry.util` named `centroid` was added which will calculate the centroid of a collection of geometric entities. And the `polygon` module now allows triangles to be instantiated from combinations of side lengths and angles (using keywords `sss`, `asa`, `sas`) and defines utility functions to convert between degrees and radians.
- In `ntheory.modular` there is a function (`solve_congruence`) to solve congruences such as “What number is 2 mod 3, 3 mod 5 and 2 mod 7?”
- A utility function named `find_unit` has been added to `phycis.units` that allows one to find units that match a given pattern or contain a given unit.
- There have been some additions and modifications to `Expr`’s methods:
 - Although the problem of proving that two expressions are equal is in general a difficult one (since whatever algorithm is used, there will always be an expression that will slip through the algorithm) the new method of `Expr` named `equals` will do its best to answer whether `A` equals `B`: `A.equals(B)` might given `True`, `False` or `None`.
 - `coeff` now supports a third argument `n` (which comes 2nd now, instead of right). This `n` is used to indicate the exponent on `x` which one seeks: `(x**2 + 3*x + 4).coeff(x, 1) -> 3`. This makes it possible to extract the constant term from a polynomial: `(x**2 + 3*x + 4).coeff(x, 0) -> 4`.
 - The method `round` has been added to round a SymPy expression to a given a number of decimal places (to the left or right of the decimal point).
- `divmod` is now supported for all SymPy numbers.
- In the `simplify` module, the algorithms for denesting of radicals (`sqrt_denest`) and simplifying gamma functions (in `combsimp`) has been significantly improved.
- The `mathematica`-similar `TableForm` function has been added to the `printing.tableform` module so one can easily generate tables with headings.
- The `expand` API has been updated. `expand()` now officially supports arbitrary `_eval_expand_hint()` methods on custom objects. `_eval_expand_hint()` methods are now only responsible for expanding the top-level expression. All `deep=True` related logic happens in `expand()` itself. See the docstring of `expand()` for more information and an example.
- Two options were added to `isympy` to aid in interactive usage. `isympy -a` automatically creates symbols, so that typing something like `a` will give `Symbol('a')`, even if you never typed `a = Symbol('a')` or `var('a')`. `isympy -i` automatically wraps integer literals with `Integer`, so that `1/2` will give `Rational(1, 2)` instead of `0.5`. `isympy -I` is the same as `isympy -a -i`. `isympy -I` makes `isympy` act much more like a traditional interactive computer algebra system. These both require IPython.
- The official documentation at <https://docs.sympy.org/> now includes an extension that automatically hooks the documentation examples in to [SymPy Live](#).

In addition to the more noticeable changes listed above, there have been numerous smaller additions, improvements and bug fixes in the commits in this release. See the git log for a full list of all changes. The command `git log sympy-0.7.1..sympy-0.7.2` will show all commits made between this release and the last. You can also see the issues closed since the last release [here](#).

8.9.6 SymPy 0.7.1

29 Jul 2011

Major changes

- Python 2.4 is no longer supported. SymPy will not work at all in Python 2.4. If you still need to use SymPy under Python 2.4 for some reason, you will need to use SymPy 0.7.0 or earlier.
- The Pyglet plotting library is now an (optional) external dependency. Previously, we shipped a version of Pyglet with SymPy, but this was old and buggy. The plan is to eventually make the plotting in SymPy much more modular, so that it supports many backends, but this has not been done yet. For now, still only Pyglet is directly supported. Note that Pyglet is only an optional dependency and is only needed for plotting. The rest of SymPy can still be used without any dependencies (except for Python).
- `isympy` now works with the new IPython 0.11.
- `mpmath` has been updated to 0.17. See the corresponding [mpmath release notes](#).
- Added a `Subs` object for representing unevaluated substitutions. This finally lets us represent derivatives evaluated at a point, i.e., `diff(f(x), x).subs(x, 0)` returns `Subs(Derivative(f(_x), _x), (_x,), (0,))`. This also means that SymPy can now correctly compute the chain rule when this functionality is required, such as with `f(g(x)).diff(x)`.
- Hypergeometric functions/Meijer G-Functions
 - Added classes `hyper()` and `meijerg()` to represent Hypergeometric and Meijer G-functions, respectively. They support numerical evaluation (using `mpmath`) and symbolic differentiation (not with respect to the parameters).
 - Added an algorithm for rewriting hypergeometric and meijer g-functions in terms of more familiar, named special functions. It is accessible via the function `hyperexpand()`, or also via `expand_func()`. This algorithm recognises many elementary functions, and also complete and incomplete gamma functions, `bessel` functions, and `error` functions. It can easily be extended to handle more classes of special functions.
- Sets
 - Added `FiniteSet` class to mimic python set behavior while also interacting with existing `Intervals` and `Unions`
 - `FiniteSets` and `Intervals` interact so that, for example `Interval(0, 10) - FiniteSet(0, 5)` produces $(0, 5) \cup (5, 10]$
 - `FiniteSets` also handle non-numerical objects so the following is possible `{1, 2, 'one', 'two', {a, b}}`
 - Added `ProductSet` to handle Cartesian products of sets
 - Create using the `*` operator, i.e. `twodice = FiniteSet(1, 2, 3, 4, 5, 6) * FiniteSet(1, 2, 3, 4, 5, 6)` or `square = Interval(0, 1) * Interval(0, 1)`
 - `pow` operator also works as expected: `R3 = Interval(-oo, oo)**3 ; (3, -5, 0) in R3 == True`
 - Subtraction, union, measurement all work taking complex intersections into account.

- Added `as_relational` method to sets, producing boolean statements using `And`, `Or`, `Eq`, `Lt`, `Gt`, etc.
- Changed `reduce_poly_inequalities` to return unions of sets rather than lists of sets
- Iterables
 - Added generating routines for integer partitions and binary partitions. The routine for integer partitions takes 3 arguments, the number itself, the maximum possible element allowed in the partitions generated and the maximum possible number of elements that will be in the partition. Binary partitions are characterized by containing only powers of two.
 - Added generating routine for multi-set partitions. Given a multiset, the algorithm implemented will generate all possible partitions of that multi-set.
 - Added generating routines for bell permutations, derangements, and involutions. A bell permutation is one in which the cycles that compose it consist of integers in a decreasing order. A derangement is a permutation such that the *i*th element is not at the *i*th position. An involution is a permutation that when multiplied by itself gives the identity permutation.
 - Added generating routine for unrestricted necklaces. An unrestricted necklace is an *a*-ary string of *n* characters, each of a possible types. These have been characterized by the parameters *n* and *k* in the routine.
 - Added generating routine for oriented forests. This is an implementation of algorithm S in TAOCP Vol 4A.
- xyz Spin bases
 - The `represent`, `rewrite` and `InnerProduct` logic has been improved to work between any two spin bases. This was done by utilizing the Wigner-D matrix, implemented in the `WignerD` class, in defining the changes between the various bases. Representing a state, i.e. `represent(JzKet(1,0), basis=Jx)`, can be used to give the vector representation of any get in any of the x/y/z bases for numerical values of *j* and *m* in the spin eigenstate. Similarly, rewriting states into different bases, i.e. `JzKet(1,0).rewrite('Jx')`, will write the states as a linear combination of elements of the given basis. Because this relies on the `represent` function, this only works for numerical *j* and *m* values. The inner product of two eigenstates in different bases can be evaluated, i.e. `InnerProduct(JzKet(1,0), JxKet(1,1))`. When two different bases are used, one state is rewritten into the other basis, so this requires numerical values of *j* and *m*, but innerproducts of states in the same basis can still be done symbolically.
 - The `Rotation.D` and `Rotation.d` methods, representing the Wigner-D function and the Wigner small-d function, return an instance of the `WignerD` class, which can be evaluated with the `doit()` method to give the corresponding matrix element of the Wigner-D matrix.
- Other changes
 - We now use MathJax in our docs. MathJax renders LaTeX math entirely in the browser using Javascript. This means that the math is much more readable than the previous png math, which uses images. MathJax is only supported on modern browsers, so LaTeX math in the docs may not work on older browsers.
 - `nroots()` now lets you set the precision of computations
 - Added support for gmpy and mpmath's types to `sympify()`

- Fix some bugs with `lambdify()`
- Fix a bug with `as_independent` and non-commutative symbols.
- Fix a bug with `collect` ([sympy/sympy#5615](#))
- Many fixes relating to porting SymPy to Python 3. Thanks to our GSoC student Vladimir Perić, this task is almost completed.
- Some people were retroactively added to the AUTHORS file.
- Added a solver for a special case of the Riccati equation in the ODE module.
- Iterated derivatives are pretty printed in a concise way.
- Fix a bug with integrating functions with multiple `DiracDeltas`.
- Add support for `Matrix.norm()` that works for Matrices (not just vectors).
- Improvements to the Gröbner bases algorithm.
- `Plot.saveimage` now supports a StringIO outfile
- `Expr.as_ordered_terms` now supports non lex orderings.
- `diff` now canonicalizes the order of differentiation symbols. This is so it can simplify expressions like `f(x, y).diff(x, y) - f(x, y).diff(y, x)`. If you want to create a `Derivative` object without sorting the args, you should create it explicitly with `Derivative`, so that you will get `Derivative(f(x, y), x, y) != Derivative(f(x, y), y, x)`. Note that internally, derivatives that can be computed are always computed in the order that they are given in.
- Added functions `is_sequence()` and `iterable()` for determining if something is an ordered iterable or normal iterable, respectively.
- Enabled an option in Sphinx that adds a source link next to each function, which links to a copy of the source code for that function.

In addition to the more noticeable changes listed above, there have been numerous other smaller additions, improvements and bug fixes in the ~300 commits in this release. See the git log for a full list of all changes. The command `git log sympy-0.7.0..sympy-0.7.1` will show all commits made between this release and the last. You can also see the issues closed since the last release [here](#).

8.9.7 SymPy 0.7.0

28 Jun 2011

Major Changes

- Polys
 - New internal representations of dense and sparse polynomials (see [6aecdb7](#), [31c9aa4](#))
 - Implemented algorithms for real and complex root isolation and counting (see [3acac67](#), [4b75dae](#), [fa1206e](#), [103b928](#), [45c9b22](#), [8870c8b](#), [b348b30](#))
 - Improved Gröbner bases algorithm (see [ff65e9f](#), [891e4de](#), [310a585](#))
 - Field isomorphism algorithm (see [b097b01](#), [08482bf](#))

- Implemented efficient orthogonal polynomials (see [b8fbd59](#))
 - Added configuration framework for polys (see [33d8cdb](#), [7eb81c9](#))
 - Function for computing minimal polynomials (see [88bf187](#), [f800f95](#))
 - Function for generating Viète's formulas (see [1027408](#))
 - `roots()` supports more classes of polynomials (e.g. cyclotomic) (see [d8c8768](#), [75c8d2d](#))
 - Added a function for recognizing cyclotomic polynomials (see [b9c2a9a](#))
 - Added a function for computing Horner form of polynomials (see [8d235c7](#))
 - Added a function for computing symmetric reductions of polynomials (see [6d560f3](#))
 - Added generators of Swinnerton-Dyer, cyclotomic, symmetric, random and interpolating polynomials (see [dad03dd](#), [6ccf20c](#), [dc728d6](#), [2f17684](#), [3004db8](#))
 - Added a function computing isolation intervals of algebraic numbers (see [37a58f1](#))
 - Polynomial division (`div()`, `rem()`, `quo()`) now defaults to a field (see [a72d188](#))
 - Added wrappers for numerical root finding algorithms (see [0d98945](#), [f638fcf](#))
 - Added symbolic capabilities to `factor()`, `sqf()` and related functions (see [d521c7f](#), [548120b](#), [f6f74e6](#), [b1c49cd](#), [3527b64](#))
 - `together()` was significantly improved (see [dc327fe](#))
 - Added support for iterable containers to `gcd()` and `lcm()` (see [e920870](#))
 - Added a function for constructing domains from coefficient containers (see [a8f20e6](#))
 - Implemented greatest factorial factorization (see [d4dbbb5](#))
 - Added partial fraction decomposition algorithm based on undetermined coefficient approach (see [9769d49](#), [496f08f](#))
 - `RootOf` and `RootSum` were significantly improved (see [f3e432](#), [4c88be6](#), [41502d7](#))
 - Added support for gmpy (GNU Multiple Precision Arithmetic Library) (see [38e1683](#))
 - Allow to compile `sympy.polys` with Cython (see [afb3886](#))
 - Improved configuration of variables in `Poly` (see [22c4061](#))
 - Added documentation based on Wester's examples (see [1c23792](#))
 - Irreducibility testing over finite fields (see [17e8f1f](#))
 - Allow symmetric and non-symmetric representations over finite fields (see [60fbff4](#))
 - More consistent factorization forms from `factor()` and `sqf()` (see [5df77f5](#))
 - Added support for automatic recognition algebraic extensions (see [7de602c](#))
 - Implemented Collins' modular algorithm for computing resultants (see [950969b](#))
 - Implemented Berlekamp's algorithm for factorization over finite fields (see [70353e9](#))
 - Implemented Trager's algorithm for factorization over algebraic number fields (see [bd0be06](#))
 - Improved Wang's algorithm for efficient factorization of multivariate polynomials (see [425e225](#))
- Quantum

- Symbolic, abstract dirac notation in `sympy.physics.quantum`. This includes operators, states (bras and kets), commutators, anticommutators, dagger, inner products, outer products, tensor products and Hilbert spaces
- Symbolic quantum computing framework that is based on the general capabilities in `sympy.physics.quantum`. This includes qubits (`sympy.physics.quantum.qubit`), gates (`sympy.physics.quantum.gate`), Grover's algorithm (`sympy.physics.quantum.grover`), the quantum Fourier transform (`sympy.physics.quantum.qft`), Shor's algorithm (`sympy.physics.quantum.shor`) and circuit plotting (`sympy.physics.quantum.circuitplot`)
- Second quantization framework that includes creation/annihilation operators for both Fermions and Bosons and Wick's theorem for Fermions (`sympy.physics.secondquant`).
- Symbolic quantum angular momentum (spin) algebra (`sympy.physics.quantum.spin`)
- Hydrogen wave functions (Schroedinger) and energies (both Schroedinger and Dirac)
- Wave functions and energies for 1D harmonic oscillator
- Wave functions and energies for 3D spherically symmetric harmonic oscillator
- Wigner and Clebsch Gordan coefficients
- Everything else
 - Implement `sympy.array`, providing numpy nd-arrays of symbols.
 - update `mpmath` to 0.16
 - Add a tensor module (see [this report](#))
 - A lot of stuff was being imported with `from sympy import *` that shouldn't have been (like `sys`). This has been fixed.
- Assumptions:
 - Refine
 - Added predicates (see [7c0b857](#), [53f0e1a](#), [d1dd6a3](#))
 - Added query handlers for algebraic numbers (see [f3bee7a](#))
 - Implement a SAT solver (see [this](#), [2d96329](#), [acfbe75](#), etc)
- Concrete
 - Finalized implementation of Gosper's algorithm (see [0f187e5](#), [5888024](#))
 - Removed redundant `Sum2` and related classes (see [ef1f6a7](#))
- Core:
 - Split `Atom` into `Atom` and `AtomicExpr` (see [965aa91](#))
 - Various `sympify()` improvements
 - Added functionality for action verbs (many functions can be called both as global functions and as methods e.g. `a.simplify() == simplify(a)`)
 - Improve handling of rational strings (see [053a045](#), [sympy/sympy#4877](#))
 - Major changes to factoring of integers (see [273f450](#), [sympy/sympy#5102](#))

- Optimized `.has()` (see [c83c9b0](#), [sympy/sympy#5079](#), [d86d08f](#))
- Improvements to power (see [c8661ef](#), [sympy/sympy#5062](#))
- Added range and lexicographic syntax to `symbols()` and `var()` (see [f6452a8](#), [9aeb220](#), [957745a](#))
- Added modulus argument to `expand()` (see [1ea5be8](#))
- Allow to convert `Interval` to relational form (see [4c269fe](#))
- SymPy won't manipulate minus sign of expressions any more (see [6a26941](#), [9c6bf0f](#), [e9f4a0a](#))
- `Real` and `.is_Real` were renamed to `Float` and `.is_Float`. `Real` and `.is_Real` still remain as deprecated shortcuts to `Float` and `is_Float` for backwards compatibility. (see [abe1c49](#))
- Methods `coeff` and `as_coefficient` are now non-commutative aware. (see [a4ea170](#))
- Geometry:
 - Various improvements to `Ellipse`
 - Updated documentation to numpy standard
 - `Polygon` and `Line` improvements
 - Allow all geometry objects to accept a tuple as `Point` args
- Integrals:
 - Various improvements (see e.g. [sympy/sympy#4871](#), [sympy/sympy#5098](#), [sympy/sympy#5091](#), [sympy/sympy#5086](#))
- isympy
 - Fixed the `-p` switch (see [e8cb04a](#))
 - Caching can be disabled using `-C` switch (see [0d8d748](#))
 - Ground types can be set using `-t` switch (see [75734f8](#))
 - Printing ordering can be set using `-o` switch (see [fcc6b13](#), [4ec9dc5](#))
- Logic
 - `implies` object adheres to negative normal form
 - Create new boolean class, `logic.boolalg.Boolean`
 - Added XOR operator (`^`) support
 - Added If-then-else (ITE) support
 - Added the `dp11` algorithm
- Functions:
 - Added `Piecewise`, `B-splines`
 - Spherical Bessel function of the second kind implemented
 - Add series expansions of multivariate functions (see [d4d351d](#))
- Matrices:
 - Add elementwise product (Hadamard product)
 - Extended QR factorization for general full ranked `m`x`n` matrices

- Remove deprecated functions `zero()`, `zeronm()`, `one()` (see [5da0884](#))
- Added cholesky and LDL factorizations, and respective solves.
- Added functions for efficient triangular and diagonal solves.
- `SMatrix` was renamed to `SparseMatrix` (see [acd1685](#))
- Printing:
 - Implemented pretty printing of binomials (see [58c1dad](#))
 - Implemented pretty printing of `Sum()` (see [84f2c22](#), [95b4321](#))
 - `sympy.printing` now supports ordering of terms and factors (see [859bb33](#))
 - Lexicographic order is now the default. Now finally things will print as $x^2 + x + 1$ instead of $1 + x + x^2$, however series still print using reversed ordering, e.g. $x - x^3/6 + O(x^5)$. You can get the old order (and other orderings) by setting the `-o` option to `isympy` (see [08b4932](#), [a30c5a3](#))
- Series:
 - Implement a function to calculate residues, `residue()`
 - Implement `nseries` and `lseries` to handle $x \neq 0$, series should be more robust now (see [2c99999](#), [sympy/sympy#5221](#) - [sympy/sympy#5223](#))
 - Improvements to Gruntz algorithm
- Simplify:
 - Added `use()` (see [147c142](#))
 - `ratsimp()` now uses `cancel()` and `reduced()` (see [108fb41](#))
 - Implemented `EPath` (see [696139d](#), [bf90689](#))
 - a new keyword `rational` was added to `nsimplify` which will replace Floats with Rational approximations. (see [053a045](#))
- Solvers:
 - ODE improvements (see [d12a2aa](#), [3542041](#); [73fb9ac](#))
 - Added support for solving inequalities (see [328eaba](#), [8455147](#), [f8fcaa7](#))
- Utilities:
 - Improve `cartes`, for generating the Cartesian product (see [b1b10ed](#))
 - Added a function computing topological sort of graphs (see [b2ce27b](#))
 - Allow to setup a customized printer in `lambdify()` (see [c1ad905](#))
 - `flatten()` was significantly improved (see [31ed8d7](#))
 - Major improvements to the Fortran code generator (see [report](#), [3383aa3](#), [7ab2da2](#), etc)

Compatibility breaks

- This will be the last release of SymPy to support Python 2.4. Dropping support for Python 2.4 will let us move forward with things like supporting Python 3, and will let us use things that were introduced in Python 2.5, like with-statement context managers.
- no longer support creating matrices without brackets (see [sympy/sympy#4029](#))
- Renamed `sum()` to `summation()` (see [3e763a8](#), [sympy/sympy#4475](#), [sympy/sympy#4826](#)). This was changed so that it no longer overrides the built-in `sum()`. The unevaluated summation is still called `Sum()`.
- Renamed `abs()` to `Abs()` (see [64a12a4](#), [sympy/sympy#4826](#)). This was also changed so that it no longer overrides the built-in `abs()`. Note that because of `__abs__` magic, you can still do `abs(expr)` with the built-in `abs()`, and it will return `Abs(expr)`.
- Renamed `max_()` and `min_()` to now `Max()` and `Min()` (see [99a271e](#), [sympy/sympy#5252](#))
- Changed behaviour of `symbols()`. `symbols('xyz')` gives now a single symbol ('xyz'), not three ('x', 'y' and 'z') (see [f6452a8](#)). Use `symbols('x,y,z')` or `symbols('x y z')` to get three symbols. The `each_char` option will still work but is being deprecated.
- Split class `Basic` into new classes `Expr`, `Boolean` (see [a0ab479](#), [635d89c](#)). Classes that are designed to be part of standard symbolic expressions (like `x**2*sin(x)`) should subclass from `Expr`. More generic objects that do not work in symbolic expressions but still want the basic SymPy structure like `.args` and basic methods like `.subs()` should only subclass from `Basic`.
- `as_basic()` method was renamed to `as_expr()` to reflect changes in the core (see [e61819d](#), [80dfe91](#))
- Methods `as_coeff_terms` and `as_coeff_factors` were renamed to `as_coeff_mul` and `as_coeff_add`, respectively.
- Removed the `trim()` function. The function is redundant with the new polys. Use the `cancel()` function instead.
- The `assume_pos_real` option to `logcombine()` was renamed to `force` to be consistent with similar force options to other functions.

In addition to the more noticeable changes listed above, there have been numerous other smaller additions, improvements and bug fixes in the ~2000 commits in this release. See the [git log](#) for a full list of all changes. The command `git log sympy-0.6.7..sympy-0.7.0` will show all commits made between this release and the last. You can also see the issues closed since the last release [here](#).

8.9.8 SymPy 0.6.7

17 Mar 2010

- fix a bug where `bin/test` and `bin/doctest` would be installed into `/usr/bin`
- fix an example for recent matplotlib versions
- some fixes for Python 2.4 and 2.5
- try to expand integrand if integration fails
- improved `isprime()` (pseudoprimes were falsely reported as being prime)

- runtests.py prints less when testing documentation
- ODE code clean-up and fixes
- runtests.py is now more verbose about expected failures to avoid confusion
- update mpmath to version 0.14
- improvements to second quantization module and coupled cluster example
- implement `visual.factorint()`
- implement `symparray()`: numpy array of sympy symbols
- documentation fixes and some bugfixes

8.9.9 SymPy 0.6.6

20 Dec 2009

- many documentation improvements, including docstrings and doctests
- new assumptions system (See assumptions documentation for more information or have a look at Fabian's blog.)
- improvements to test runner
- printing improvements (especially LaTeX, but also mathml and pretty printing)
- discriminant of polys
- block diagonal methods for matrices
- vast improvements to solving of ODEs (See ODE documentation for full details or Aaron's blog).
- logcombine function
- improvements to sets
- better trigonometric simplification
- improvements to piecewise functions
- improvements to `solve()` and `nsolve()`
- improvements to `as_numer_denom()`
- much better quartic and cubic polynomial rootfinding
- code refactoring and cleanup
- physics: coupled clusters and wick expansion
- matrices: symbolic QR solving
- mpmath updated
- pyglet updated
- many, many bug fixes and small improvements

8.9.10 SymPy 0.6.5

17 Jul 2009

- Geometric Algebra Improvements
 - Upgrade GA module with left and right contraction operations
 - Add intersection test for the vertical segment, reimplement of `convex_hull`
- Implement `series()` as function
- Core improvements
 - Refactor `Number.eval_power`
 - fix bugs in `Number.eval_power`
- Matrix improvements:
 - Improve jacobian function, introduce `vec` and `vech`
- Solver improvements:
 - solutions past linear factor found in `tsolve`
 - refactor `sympy.solvers.guess_solve_strategy`
 - Small cleanups to the ODE solver and tests
 - Fix corner case for Bernoulli equation
- Improvements on partial differential equations solvers
 - Added separation of variables for PDEs
- Expand improvements
 - Refactoring
 - `exp(x) exp(y)` is no longer automatically combined into `exp(x+y)`, use `powsimp` for that
- Documentation improvements:
 - Test also documentation under `doc/`
 - Added many docstrings
 - Fix Sphinx complaints/warnings/errors
 - Doctest coverage
- New logic module
 - Efficient DPLL algorithm
- LaTeX printer improvements:
 - Handle standard form in the LaTeX printer correctly
 - Latex: `print_Mul` fix ([sympy/sympy#4381](#))
 - Robust printing of latex sub and superscripts
 - sorting `print_Add` output using a main variable
 - Matrix printing improvements
- MathML printing improvements:

- MathML's printer extended
- Testing framework improvements
 - Make tests pass without the “py” module
- Polynomial module improvements:
 - Fixed subresultant PRS computation and `ratint()`
 - Removed old module `sympy.polynomials`
- limit fixes:
 - Compute the finite parts of the limit of a sum by direct substitution
- Test coverage script
- Code quality improvements (remove string exceptions, code quality test improvements)
- C code generation
- Update mpmath

8.9.11 SymPy 0.6.4

4 Apr 2009

- robust and fast (still pure Python) multivariate factorization
- sympy works with pickle protocol 2 (thus works in ipython parallel)
- `./sympy` test now uses our testing suite and it tests both regular tests and doctests
- examples directory tidied up
- more trigonometric simplifications
- polynomial roots finding and factoring vastly improved
- mpmath updated
- many bugfixes (more than 200 patches since the last release)

8.9.12 SymPy 0.6.3

19 Nov 2008

- port to python2.6 (all tests pass)
- port to jython (all tests pass except those depending on the “ast” module)
- true division fixed (all tests pass with “-Qnew” Python option)
- <http://buildbot.sympy.org> created, sympy is now regularly tested on python2.4, 2.5, 2.6 on both i386 and amd64 architectures.
- `py.bench` – `py.test` based benchmarking added
- `bin/test` – simple `py.test` like testing framework, without external dependencies, nice colored output
- most limits now work
- factorization over $Z[x]$ greatly improved

- Piecewise function added
- `nsimplify()` implemented
- symbols and var syntax unified
- C code printing
- many bugfixes

8.9.13 SymPy 0.6.2

17 Aug 2008

- SymPy is now 50% faster on average (cache:on) and 130% (cache:off) compared to previous release.
- `adaptive` and faster `evalf()`
- `evalf`: numerical summation of hypergeometric series
- `evalf`: fast and accurate numerical summation
- `evalf`: oscillatory quadrature
- integrals now support variable transformation
- we can now `integrate(f(x)·diff(f(x),x), x)`
- we can now solve $a \cdot \cos(x) = y$ and $\exp(x) + \exp(-x) = y$
- printing system refactored
- `pprint`: new symbol for multiply in unicode mode ($x*y \rightarrow x \cdot y$)
- `pprint`: matrices now look much better
- printing of dicts and sets are now more human-friendly
- `latex`: now supports sub- and superscripts in symbol names
- `RootSum.doit()`, now works on all roots
- Wild can now have additional predicates
- numpy-like zeros and ones functions
- `var('x,y,z')` now works
- `((x+y+z)**50).expand()` is now 4.8x faster
- big assumptions cleanup and rewrite
- access to all object attributes is now ~ 2.5 times faster
- we try not to let 'is_commutative' to go through (slow) assumptions path
- Add/Mul were optimized (for some cases significantly)
- `isympy` and `sympy.interactive` code were merged
- multiple inheritance removed (NoArithMeths, NoRelMeths, RelMeths, ArithMeths are gone)
- `.nseries()` is now used as default in `.series()`
- doctesting was made more robust

8.9.14 SymPy 0.6.1

22 Jul 2008

- almost all functions and constants can be converted to Sage
- univariate factorization algorithm was fixed
- `.evalf()` method fixed, `pi.evalf(106)` calculates 1 000 000 digits of pi
- `@threaded` decorator
- more robust solvers, polynomials and simplification
- better simplify, that makes a solver more robust
- optional compiling of functions to machine code
- `msolve`: solving of nonlinear equation systems using Newton's method
- `((x+y+z)**50).expand()` is now 3 times faster
- caching was removed from the Order class: 1.5x speedups in series tests

8.9.15 SymPy 0.6.0

7 Jul 2008

- all documentation wiki pages moved to <https://docs.sympy.org/>
- `mpmath` was integrated in SymPy, `numerics` module removed
- `mpmath` can use `gmpy` optionally, thus calculating 1000000 digits of pi in 7.5s
- Common subexpression elimination implemented
- `roots`, `RootsOf`, `RootSum` implemented
- `lambdify()` now accepts Matrices
- Matrices polished and spedup
- `source` command implemented
- Polys were made the default polynomials in SymPy
- `Add`, `Mul`, `Pow` now accept `evaluate=False` argument

8.9.16 SymPy 0.5.15

24 May 2008

- all SymPy functions support vector arguments, e.g. `sin([1, 2, 3])`
- `lambdify` can now use `numpy/math/mpmath`
- the order of `lambdify` arguments has changed
- all SymPy objects are pickable
- `simplify` improved and made more robust
- `broken_limit_series` was removed, we now have just one limit implementation
- limits now use `.nseries()`

- `.nseries()` improved a lot
- Polys improved
- Basic kronecker delta and Levi-Civita implementation

8.9.17 SymPy 0.5.14

26 Apr 2008

- SymPy is now 25% faster on average compared to the previous release
 - `__eq__`/`__ne__`/`__nonzero__` returns True/False directly so dict lookups are not expensive anymore
 - `sum(x**i/i, i=1..400)` is now 4.8x faster
 - `isinstance(term, C.Mul)` was replaced by `term.is_Mul` and similarly for other basic classes
- Documentation was improved a lot. See <https://docs.sympy.org/>
- `rsolve_poly` & `rsolve_hyper` fixed
- `subs` and `subs_dict` unified to `.subs()`
- faster and more robust polynomials module
- improved `Matrix.det()`, implemented Berkowitz algorithm
- improved `isympy` (interactive shell for SymPy)
- pretty-printing improved
- `Rel`, `Eq`, `Ne`, `Lt`, `Le`, `Gt`, `Ge` implemented
- `Limit` class represents unevaluated limits now
- Bailey-Borwein-Plouffe algorithm (finds the nth hexadecimal digit of pi without calculating the previous digits) implemented
- solver for transcendental equations added
- `.nseries()` methods implemented (more robust/faster than `.oseries`)
- multivariate Lambdas implemented

8.9.18 SymPy 0.5.13

6 Mar 2008

- SymPy is now 2x faster in average compared to the previous release
 - first patches with 25% speedup
 - `Basic.cos` et. al. removed, use `C.cos` instead
 - `sympy.core` now uses direct imports
 - `sympifyit` decorator
 - speedup Integers creation and arithmetic
 - speedup unary operations for singleton numbers

- remove silly slowdowns from fast-path of mul and div
- significant speedup was achieved by reusing dummy variables
- `is_dummy` is not an assumption anymore
- Symbols & Wilds are cached
- `((2+3*I)**1000).expand()` is now at least 100x faster
- `.expand()` was made faster for cases where an expression is already expanded
- rational powers of integers are now computed more efficiently
- unknown assumptions are now cached as well as known assumptions
- `integrate()` can handle most of the basic integrals now
- interactive experience with isympy was improved through adding support for `,` `()` and `{ }` to pretty-printer, and switching to it as the default ipython printer
- new `trim()` function to map all non-atomic expressions, ie. functions, derivatives and more complex objects, to symbols and remove common factors from numerator and denominator. also `cancel()` was improved
- `.expand()` for noncommutative symbols fixed
- bug in `(x+y+sin(x)).as_independent()` fixed
- `.subs_dict()` improved
- support for plotting geometry objects added
- bug in `.tangent_line()` of ellipse fixed
- new `atan2` function and associated fixes for `.arg()` and expanding rational powers
- new `.coeff()` method for returning coefficient of a poly
- pretty-printer now uses unicode by default
- recognition of geometric sums were generalized
- `.is_positive` and `.is_negative` now fallback to `evalf()` when appropriate
- as the result `oo*(pi-1)` now correctly simplifies to `oo`
- support for objects which provide `__int__` method was added
- we finally started SymPy User's Guide
- `BasicMeths` merged into `Basic`
- cache subsystem was cleaned up - now it supports only immutable objects

8.9.19 SymPy 0.5.12

27 Jan 2008

- SymPy works with NumPy out of the box.
- `RootOf` implemented.
- Lambda support works now.
- Heuristic Risch method improved.
- `cancel()` function implemented.

- `sqrt(x)` is now equivalent to `x**(1/2)`.
- `Derivative` is now unevaluated.
- `list2numpy()` implemented.
- Series expansion of hyperbolic functions fixed.
- `sympify('lambda x: 2*x')` works, plus other fixes.
- Simple maxima parser implemented.
- `sin(x)[0]` idiom changed to `sin(x).args[0]`
- `sin(x).series(x, 5)` idiom changed to `sin(x).series(x, 0, 5)`
- Caching refactored.
- Integration of trigonometry expressions improved
- Pretty-printing for list and tuples implemented.
- Python printing implemented.
- 2D plots now don't rotate in 3D, but translate instead.

8.9.20 SymPy 0.5.11

7 Jan 2008

- `./setup.py install` installs `pyglet` correctly now.
- `var("k")` fixed.
- Script for automatic testing of plotting in pure environment added.

8.9.21 SymPy 0.5.10

4 Jan 2008

- `view` renamed to `preview`, `pngview`, `pdfview`, `dviview` added.
- Latex printer was rewritten, `preview` uses builtin `pyglet`.
- Square root denesting implemented.
- Parser of simple Mathematica expressions added.
- TeXmacs interface written.
- Some integration fixes.
- Line width in 2D plotting can be specified.
- README was updated.
- `pyglet` and `mpmath` were updated and moved to `sympy/thirdparty`
- All `sys.path` hacks were moved to just 2 places.
- SymPy objects should work in numpy arrays now.
- Hand written `sympify()` parser was rewritten and simplified using Python AST.

8.9.22 SymPy 0.5.9

22 Dec 2007

- Differential solvers were polished.
- `isympy` now predefines `f` as a function.
- Matrix printing improved.
- Printing internals were documented.

8.9.23 SymPy 0.5.8

6 Dec 2007

- `_eval_apply()` method was renamed to `canonize()`.
- Added `var` from SAGE.
- Added more number theory functions.
- Spherical harmonics (`Ylm`) implemented.
- Functions interface simplified (`SingleValuedFunction` removed, `nofargs` -> `nargs`).
- Draw negative powers in denominator nicely.
- Integration of polynomials is 10x faster.
- `pyglet` updated to 1.0beta2.

8.9.24 SymPy 0.5.7

17 Nov 2007

- `isympy` now uses 2D unicode pretty-printing by default.
- Convergence acceleration / extrapolation methods for series and sequences.
- SymPy was made ready to work nicely with SAGE.

8.9.25 SymPy 0.5.6

30 Oct 2007

- `_sage_()` methods implemented to convert any SymPy expression to a SAGE expression.
- `isympy` fixed so that it always tries the local unpacked `sympy` first (the one in the directory where `isympy` sits) and only then the system wide installation of `sympy` (Debian package for example).

8.9.26 SymPy 0.5.5

20 Oct 2007

- `sympy.abc` module for quickly importing predefined symbols.
- Nice pretty printing when a unicode terminal is available.
- `isympy -c python` now also supports true division.
- Documentation improved (sympy module, bin/isympy and it's man page).
- A lot of problems with series expansion fixed.
- Patched pyglet to conform to Debian policy.

8.9.27 SymPy 0.5.4

5 Oct 2007

- `Log` and `ApplyLog` classes were simplified to `log`, as was in the 0.4.3 version (the same for all other classes, like `sin` or `cos`).
- Limits algorithm was fixed and it works very reliably (there are some bugs in the series facility though that make some limits fail), see [this post](#) for more details.
- All functions arguments are now accessed using the `sin(x)[:]` idiom again, as in the 0.4.3 version (instead of the old `sin(x)._args` or `sin(x).args` which was briefly introduced in the 0.5.x series).

8.9.28 SymPy 0.5.3

8 Sep 2007

- Faster import sympy statement.
- Using the `integrate(3*t**2, (t, 0, x))` syntax again (as was in the 0.4.3 version).
- Using true division in isympy (`1/2` returns `0.5` instead of `0`, example).
- Plotting module can save images.
- Implemented extended Risch-Norman heuristic.
- Full partial fraction decomposition via `apart()`.
- Added a complete set of rewrite rules for trigonometric and hyperbolic functions.
- `ComplexInfinity` renamed to `zoo`.

8.9.29 SymPy 0.5.2

20 Aug 2007

- concrete mathematics module written
- geometry module
- make the tarball conform to Debian policy
- many small bugs fixed

8.9.30 SymPy 0.5.1

12 Aug 2007

- importing sympy (import sympy) was made a lot faster (2.4s against 0.1s)

8.9.31 SymPy 0.5.0

12 Aug 2007

- New core (from 10x to 100x speedup compared to 0.4.3).
- Multivariate functions.
- Pattern matching uses the Wild and WildFunction classes.
- Numerics module for fast arbitrary-precision numerical computations.
- Plotting module improved (colormaps, middle mouse button for zooming and more).
- sympy/modules/* was moved to sympy/* and the sympy/modules directory was deleted.

BIBLIOGRAPHY

- [APPM90] Yu. A. Brychkov A. P. Prudnikov and O. I. Marichev. *More Special Functions*. Volume 3 of Integrals and Series. Gordon and Breach, 1990.
- [Abr71] S.A. Abramov. On the summation of rational functions. *USSR Computational Mathematics and Mathematical Physics*, 11(4):324-330, 1971. URL: <https://www.sciencedirect.com/science/article/abs/pii/0041555371900280>, doi:10.1016/0041-5553(71)90028-0.
- [Abr95] Sergei A. Abramov. Rational Solutions of Linear Difference and q-difference Equations with Polynomial Coefficients. In *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, 285-289. New York, NY, USA, 1995. ACM Press. doi:10.1145/220346.220383.
- [ABPetkovvsek95] Sergei A. Abramov, Manuel Bronstein, and Marko Petkovšek. On Polynomial Solutions of Linear Operator Equations. In *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, 290-296. New York, NY, USA, 1995. ACM Press. doi:10.1145/220346.220384.
- [AL94] William Wells Adams and Philippe Loustau. *An Introduction to Gröbner Bases*. American Mathematical Society, Boston, MA, USA, July 1994. ISBN 0-821-83804-0.
- [ALW95] Iyad A. Ajwa, Zhuojun Liu, and Paul S. Wang. Gröbner Bases Algorithm. Technical Report ICM-199502-00, ICM Technical Reports Series, 1995.
- [ARW96] Steven Arno, M.L. Robinson, and Ferrell S. Wheeler. On denominators of algebraic numbers and integer polynomials. *Journal of Number Theory*, 57(2):292-302, 1996. URL: <https://www.sciencedirect.com/science/article/pii/S0022314X96900499>, doi:10.1006/jnth.1996.0049.
- [BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Volume 141 of Graduate Texts in Mathematics. Springer-Verlag, New York, NY, USA, 1993. ISBN 0-387-97971-9. In Cooperation with Heinz Kredel.
- [Bro] Manuel Bronstein. Poor Man's Integrator. URL: <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint/index.htm>.
- [Bro05] Manuel Bronstein. *Symbolic Integration I: Transcendental Functions*. Springer-Verlag, New York, NY, USA, second edition, 2005. ISBN 3-540-21493-3.
- [BS93] Manuel Bronstein and Bruno Salvy. Full Partial Fraction Decomposition of Rational Functions. In *ISSAC '93: Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation*, 157-160. New York, NY, USA, 1993. ACM Press. doi:10.1145/164081.164114.

- [Bro71] W. S. Brown. On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors. In *SYMSAC '71: Proceedings of the second ACM Symposium on Symbolic and Algebraic Computation*, 195–211. New York, NY, USA, 1971. ACM Press. doi:10.1145/800204.806288.
- [Bro78] W. S. Brown. The subresultant prs algorithm. *ACM Transactions on Mathematical Software*, 4(3):237–249, September 1978. URL: <https://dl.acm.org/doi/10.1145/355791.355795>, doi:10.1145/355791.355795.
- [BT71] W. S. Brown and J. F. Traub. On Euclid's Algorithm and the Theory of Subresultants. *Journal of the ACM*, 18(4):505–514, 1971. doi:10.1145/321662.321665.
- [Buc01] Bruno Buchberger. Gröbner Bases: A Short Introduction for Systems Theorists. In *Computer Aided Systems Theory — EUROCAST 2001-Revised Papers*, 1–19. London, UK, 2001. Springer-Verlag.
- [Col67] George E. Collins. Subresultants and reduced polynomial remainder sequences. *Journal of the ACM*, 14(1):128–142, January 1967. URL: <https://dl.acm.org/citation.cfm?doid=321371.321381>, doi:10.1145/321371.321381.
- [CLOShea15] David Cox, John Little, and Donald O'Shea. *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, NY, USA, fourth edition, 2015. ISBN 978-3-319-16720-6.
- [DST88] J. H. Davenport, Y. Siret, and E. Tournier. *Computer algebra: systems and algorithms for algebraic computation*. Academic Press, New York, NY, USA, 1988. ISBN 0-12-204230-1.
- [FaugereGLM93] J.C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *Journal of Symbolic Computation*, 16(4):329–344, October 1993. URL: <https://www.sciencedirect.com/science/article/pii/S0747717183710515>, doi:10.1006/jsco.1993.1051.
- [GMN+91] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. “One sugar cube, please” or selection strategies in the Buchberger algorithm. In *ISSAC '91: Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, 49–54. New York, NY, USA, 1991. ACM Press. doi:10.1145/120694.120701.
- [GB73] M.E. Goldstein and W.H. Braun. *Advanced Methods for the Solution of Differential Equations*. NASA (United States. National Aeronautics and Space Administration). Scientific and Technical Information Office, National Aeronautics and Space Administration, 1973.
- [Gru96] Dominik Gruntz. *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.
- [HNorsettW14] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer Series in Computational Mathematics. Springer-Verlag, 2014. ISBN 9783642052330.
- [JM09] Seyed Mohammad Mahdi Javadi and Michael Monagan. On Factorization of Multivariate Polynomials over Algebraic Number and Function Fields. In *ISSAC '09: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, 199–206. New York, NY, USA, 2009. ACM Press. doi:10.1145/1576702.1576731.
- [Kar81] Michael Karr. Summation in Finite Terms. *Journal of the ACM*, 28(2):305–350, 1981. doi:10.1145/322248.322255.

- [Knu85] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, second edition, 1985. ISBN 0-201-03822-6.
- [Koe98] W. Koepf. *Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities*. Braunschweig, 1998.
- [KL89] Dexter Kozen and Susan Landau. Polynomial Decomposition Algorithms. *Journal of Symbolic Computation*, 7(5):445-456, 1989. doi:10.1016/S0747-7171(89)80027-6.
- [KW88] Heinz Kredel and Volker Weispfenning. Computing dimension and independent sets for polynomial ideals. *Journal of Symbolic Computation*, 6(2):231-247, 1988. URL: <https://www.sciencedirect.com/science/article/pii/S0747717188800452>, doi:10.1016/S0747-7171(88)80045-2.
- [LF95] Hsin-Chao Liao and Richard J. Fateman. Evaluation of the heuristic polynomial GCD. In *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, 240-247. New York, NY, USA, 1995. ACM Press. doi:10.1145/220346.220376.
- [Luk69] Yudell L. Luke. *The Special Functions and Their Approximations*. Volume 1. Academic Press, New York, NY, USA, 1969.
- [Man93] Yiu-Kwong Man. On computing closed forms for indefinite summations. *Journal of Symbolic Computation*, 16(4):355-376, October 1993. URL: <https://www.sciencedirect.com/science/article/pii/S0747717183710539>, doi:10.1006/jsco.1993.1053.
- [MW94] Yiu-Kwong Man and Francis J. Wright. Fast polynomial dispersion computation and its application to indefinite summation. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '94, 175-180. New York, NY, USA, 1994. ACM Press. URL: <https://dl.acm.org/citation.cfm?doid=190347.190413>, doi:10.1145/190347.190413.
- [MVV97] A. J. Menezes, O. P. C. Van, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, Boca Raton, Florida, USA, 1997.
- [MvH04] Michael Monagan and Mark van Hoeij. Algorithms for Polynomial GCD Computation over Algebraic Function Fields. In *ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, 297-304. New York, NY, USA, 2004. ACM Press. doi:10.1145/1005285.1005328.
- [MW00] Michael B. Monagan and Allan D. Wittkopf. On the design and implementation of brown's algorithm over the integers and number fields. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation*, ISSAC '00, 225-233. New York, NY, USA, 2000. ACM Press. URL: <https://dl.acm.org/citation.cfm?doid=345542.345639>, doi:10.1145/345542.345639.
- [NKBG03] B. Buchberger N.K. Bose and J.P. Guiver. *Multidimensional Systems Theory and Applications*. Springer, second edition, 2003.
- [Petkovvsek92] Marko Petkovšek. Hypergeometric Solutions of Linear Recurrences with Polynomial Coefficients. *Journal of Symbolic Computation*, 14(2-3):243-264, 1992. doi:10.1016/0747-7171(92)90038-6.
- [PetkovvsekWZ97] Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger. *A (=) B*. AK Peters, Ltd., Wellesley, MA, USA, 1997. URL: <http://sites.math.rutgers.edu/~zeilberg/AeqB.pdf>.
- [Roa96] Kelly Roach. Hypergeometric function representations. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ISSAC

- '96, 301–308. New York, NY, USA, 1996. ACM Press. URL: <https://dl.acm.org/citation.cfm?doid=236869.237088>, doi:10.1145/236869.237088.
- [Roa97] Kelly Roach. Meijer g function representations. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, ISSAC '97, 205–211. New York, NY, USA, 1997. ACM Press. URL: <https://dl.acm.org/citation.cfm?doid=258726.258784>, doi:10.1145/258726.258784.
- [Sim16] G.F. Simmons. *Differential Equations with Applications and Historical Notes, Third Edition*. Textbooks in Mathematics. CRC Press, 2016. ISBN 9781498702621.
- [SW10] Yao Sun and Dingkang Wang. A new proof of the F5 algorithm. *CoRR*, 2010. URL: <https://arxiv.org/abs/1004.0084>, arXiv:1004.0084.
- [TP63] M. Tenenbaum and H. Pollard. *Ordinary Differential Equations*. Dover Publications, 1963.
- [Tra76] Barry M. Trager. Algebraic factoring and rational function integration. In *SYMSAC '76: Proceedings of the third ACM Symposium on Symbolic and Algebraic Computation*, 219–226. New York, NY, USA, 1976. ACM Press. doi:10.1145/800205.806338.
- [vHM02] Mark van Hoeij and Michael Monagan. A Modular GCD Algorithm over Number Fields Presented with Multiple Extensions. In *ISSAC '02: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, 109–116. New York, NY, USA, 2002. ACM Press. doi:10.1145/780506.780520.
- [Wan81] Paul S. Wang. A p-adic Algorithm for Univariate Partial Fractions. In *SYM-SAC '81: Proceedings of the fourth ACM Symposium on Symbolic and Algebraic Computation*, 212–217. New York, NY, USA, 1981. ACM Press. doi:10.1145/800206.806398.
- [YNT89] Kazuhiro Yokoyama, Masayuki Noro, and Taku Takeshima. Computing primitive elements of extension fields. *Journal of Symbolic Computation*, 8(6):553–580, 1989. URL: <https://www.sciencedirect.com/science/article/pii/S0747717189800616>, doi:10.1016/S0747-7171(89)80061-6.

PYTHON MODULE INDEX

d

- diofant, 41
- diofant.calculus, 752
- diofant.calculus.gruntz, 787
- diofant.calculus.limits, 752
- diofant.calculus.optimization, 753
- diofant.calculus.order, 753
- diofant.calculus.residues, 755
- diofant.calculus.singularities, 752
- diofant.combinatorics.generators, 169
- diofant.combinatorics.graycode, 208
- diofant.combinatorics.group_constructs, 221
- diofant.combinatorics.named_groups, 212
- diofant.combinatorics.partitions, 142
- diofant.combinatorics.perm_groups, 170
- diofant.combinatorics.permutations, 147
- diofant.combinatorics.polyhedron, 196
- diofant.combinatorics.prufer, 199
- diofant.combinatorics.subsets, 202
- diofant.combinatorics.tensor_can, 223
- diofant.combinatorics.testutil, 221
- diofant.combinatorics.util, 215
- diofant.config, 41
- diofant.core, 41
- diofant.core.add, 104
- diofant.core.assumptions, 44
- diofant.core.basic, 46
- diofant.core.cache, 45
- diofant.core.compatibility, 140
- diofant.core.containers, 139
- diofant.core.core, 55
- diofant.core.evalf, 138
- diofant.core.evaluate, 56
- diofant.core.expr, 57
- diofant.core.exprtools, 141
- diofant.core.function, 121
- diofant.core.mod, 106
- diofant.core.mul, 101
- diofant.core.multidimensional, 120
- diofant.core.numbers, 85
- diofant.core.power, 99
- diofant.core.relational, 107
- diofant.core.singleton, 55
- diofant.core.symbol, 80
- diofant.core.sympify, 41
- diofant.domains, 426
- diofant.functions, 274
- diofant.functions.special.bessel, 351
- diofant.functions.special.beta_functions, 328
- diofant.functions.special.elliptic_integrals, 372
- diofant.functions.special.error_functions, 329
- diofant.functions.special.gamma_functions, 319
- diofant.functions.special.polynomials, 374
- diofant.functions.special.zeta_functions, 363
- diofant.integrals, 392
- diofant.integrals.meijerint_doc, 809
- diofant.integrals.quadrature, 411
- diofant.integrals.transforms, 393
- diofant.interactive, 566
- diofant.interactive.printing, 566
- diofant.interactive.session, 566
- diofant.logic, 417
- diofant.matrices, 432
- diofant.matrices.dense, 479
- diofant.matrices.expressions, 500
- diofant.matrices.expressions.blockmatrix, 503
- diofant.matrices.immutable, 498
- diofant.matrices.matrices, 432
- diofant.matrices.sparse, 487
- diofant.ntheory.continued_fraction, 257
- diofant.ntheory.egyptian_fraction, 259
- diofant.ntheory.factor_, 235
- diofant.ntheory.generate, 228
- diofant.ntheory.modular, 246
- diofant.ntheory.multinomial, 249
- diofant.ntheory.partitions_, 250

- diofant.ntheory.primetest, 250
- diofant.ntheory.residue_ntheory, 252
- diofant.parsing, 748
- diofant.polys, 505
- diofant.polys.constructor, 539
- diofant.polys.euclidtools, 764
- diofant.polys.factorization_alg_field, 771
- diofant.polys.factortools, 767
- diofant.polys.groebnertools, 767
- diofant.polys.modulargcd, 775
- diofant.polys.monomials, 540
- diofant.polys.numberfields, 539
- diofant.polys.orderings, 540
- diofant.polys.orthopolys, 543
- diofant.polys.partfrac, 545
- diofant.polys.polyerrors, 785
- diofant.polys.polyfuncs, 537
- diofant.polys.polyoptions, 785
- diofant.polys.polyroots, 542
- diofant.polys.polytools, 505
- diofant.polys.rationaltools, 544
- diofant.polys.rootisolation, 784
- diofant.polys.rootoftools, 540
- diofant.polys.specialpolys, 543
- diofant.polys.sqfreetools, 785
- diofant.printing, 548
- diofant.printing.ccode, 551
- diofant.printing.codeprinter, 562
- diofant.printing.conventions, 562
- diofant.printing.fcode, 554
- diofant.printing.lambdarepr, 558
- diofant.printing.latex, 558
- diofant.printing.mathematica, 557
- diofant.printing.mathml, 560
- diofant.printing.precedence, 562
- diofant.printing.pretty, 551
- diofant.printing.pretty_symbolology, 562
- diofant.printing.printer, 548
- diofant.printing.python, 561
- diofant.printing.repr, 561
- diofant.printing.str, 561
- diofant.printing.stringpict, 563
- diofant.sets.fancysets, 578
- diofant.sets.sets, 567
- diofant.simplify.combsimp, 599
- diofant.simplify.cse_main, 601
- diofant.simplify.epathtools, 604
- diofant.simplify.fu, 595
- diofant.simplify.hyperexpand, 603
- diofant.simplify.hyperexpand_doc, 797
- diofant.simplify.powsimp, 596
- diofant.simplify.radsimp, 587
- diofant.simplify.ratsimp, 593
- diofant.simplify.sqrtdenest, 599
- diofant.simplify.traversaltools, 604
- diofant.simplify.trigsimp, 593
- diofant.solvers, 606
- diofant.solvers.deutils, 696
- diofant.solvers.diophantine, 610
- diofant.solvers.inequalities, 610
- diofant.solvers.ode, 680
- diofant.solvers.pde, 695
- diofant.solvers.polysys, 608
- diofant.solvers.recurr, 684
- diofant.solvers.solvers, 606
- diofant.solvers.utils, 695
- diofant.tensor, 696
- diofant.tensor.array, 697
- diofant.tensor.index_methods, 708
- diofant.tensor.indexed, 702
- diofant.utilities, 710
- diofant.utilities.autowrap, 711
- diofant.utilities.codegen, 716
- diofant.utilities.decorator, 725
- diofant.utilities.enumerative, 725
- diofant.utilities.iterables, 731
- diofant.utilities.lambdify, 744
- diofant.utilities.memoization, 747
- diofant.utilities.misc, 747
- diofant.utilities.randtest, 747

Symbols

<code>__add__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_alpha_to_z()</code> (in module diofant.polys.factorization_alg_field), 771
<code>__contains__()</code> (diofant.combinatorics.perm_groups.PermutationGroup method), 171	<code>_base_ordering()</code> (in module diofant.combinatorics.util), 215
<code>__eq__()</code> (diofant.combinatorics.perm_groups.PermutationGroup method), 172	<code>_check_cycles_alt_sym()</code> (in module diofant.combinatorics.util), 216
<code>__eq__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_chinese_remainder_reconstruction()</code> (in module diofant.polys.modulargcd), 775
<code>__getitem__()</code> (diofant.matrices.dense.DenseMatrix method), 479	<code>_cmp_perm_lists()</code> (in module diofant.combinatorics.testutil), 221
<code>__getitem__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_denominator()</code> (in module diofant.polys.factorization_alg_field), 771
<code>__mul__()</code> (diofant.combinatorics.perm_groups.PermutationGroup method), 172	<code>_diophantine()</code> (in module diofant.polys.factorization_alg_field), 771
<code>__mul__()</code> (diofant.matrices.dense.DenseMatrix method), 480	<code>diophantine_univariate()</code> (in module diofant.polys.factorization_alg_field), 771
<code>__mul__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_distinct_prime_divisors()</code> (in module diofant.polys.factorization_alg_field), 772
<code>__new__()</code> (diofant.combinatorics.perm_groups.PermutationGroup static method), 172	<code>_distribute_gens_by_base()</code> (in module diofant.combinatorics.util), 216
<code>__pow__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_div()</code> (in module diofant.polys.modulargcd), 776
<code>__radd__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_euclidean_algorithm()</code> (in module diofant.polys.modulargcd), 776
<code>__rmul__()</code> (diofant.polys.rings.PolyElement method), 758	<code>_evaluate_ground()</code> (in module diofant.polys.modulargcd), 776
<code>__rsub__()</code> (diofant.polys.rings.PolyElement method), 759	<code>_extended_euclidean_algorithm()</code> (in module diofant.polys.factorization_alg_field), 772
<code>__setitem__()</code> (diofant.polys.rings.PolyElement method), 759	<code>_factor()</code> (in module diofant.polys.factorization_alg_field), 772
<code>__sub__()</code> (diofant.polys.rings.PolyElement method), 759	<code>_func_field_modgcd_m()</code> (in module diofant.polys.modulargcd), 776
<code>__weakref__</code> (diofant.polys.rings.PolyElement attribute), 759	<code>_func_field_modgcd_p()</code> (in module diofant.polys.modulargcd), 776
<code>_af_parity()</code> (in module diofant.combinatorics.permutations), 168	

[fant.polys.modulargcd\(\)](#), 777
[_gf_gcdex\(\)](#) (in module [diofant.polys.modulargcd](#)), 778
[_handle_Integral\(\)](#) (in module [diofant.solvers.ode](#)), 684
[_handle_precomputed_bsigs\(\)](#) (in module [diofant.combinatorics.util](#)), 217
[_hensel_lift\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 772
[_interpolate\(\)](#) (in module [diofant.polys.modulargcd](#)), 778
[_leading_coeffs\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 772
[_linear_2eq_order1_type3\(\)](#) (in module [diofant.solvers.ode](#)), 668
[_linear_2eq_order1_type4\(\)](#) (in module [diofant.solvers.ode](#)), 669
[_linear_2eq_order1_type5\(\)](#) (in module [diofant.solvers.ode](#)), 669
[_linear_2eq_order1_type6\(\)](#) (in module [diofant.solvers.ode](#)), 669
[_linear_2eq_order1_type7\(\)](#) (in module [diofant.solvers.ode](#)), 670
[_linear_2eq_order2_type1\(\)](#) (in module [diofant.solvers.ode](#)), 671
[_linear_2eq_order2_type11\(\)](#) (in module [diofant.solvers.ode](#)), 676
[_linear_2eq_order2_type2\(\)](#) (in module [diofant.solvers.ode](#)), 672
[_linear_2eq_order2_type3\(\)](#) (in module [diofant.solvers.ode](#)), 672
[_linear_2eq_order2_type5\(\)](#) (in module [diofant.solvers.ode](#)), 673
[_linear_2eq_order2_type6\(\)](#) (in module [diofant.solvers.ode](#)), 673
[_linear_2eq_order2_type7\(\)](#) (in module [diofant.solvers.ode](#)), 674
[_linear_2eq_order2_type8\(\)](#) (in module [diofant.solvers.ode](#)), 674
[_linear_2eq_order2_type9\(\)](#) (in module [diofant.solvers.ode](#)), 675
[_linear_3eq_order1_type4\(\)](#) (in module [diofant.solvers.ode](#)), 676
[_minpoly_from_dense\(\)](#) (in module [diofant.polys.modulargcd](#)), 778
[_modgcd_p\(\)](#) (in module [diofant.polys.modulargcd](#)), 778
[_monic_associate\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 773
[_naive_list_centralizer\(\)](#) (in module [diofant.combinatorics.testutil](#)), 221
[_nonlinear_2eq_order1_type1\(\)](#) (in module [diofant.solvers.ode](#)), 677
[_nonlinear_2eq_order1_type2\(\)](#) (in module [diofant.solvers.ode](#)), 677
[_nonlinear_2eq_order1_type3\(\)](#) (in module [diofant.solvers.ode](#)), 678
[_nonlinear_2eq_order1_type4\(\)](#) (in module [diofant.solvers.ode](#)), 678
[_nonlinear_2eq_order1_type5\(\)](#) (in module [diofant.solvers.ode](#)), 679
[_nonlinear_3eq_order1_type1\(\)](#) (in module [diofant.solvers.ode](#)), 679
[_nonlinear_3eq_order1_type2\(\)](#) (in module [diofant.solvers.ode](#)), 680
[_orbits_transversals_from_bsigs\(\)](#) (in module [diofant.combinatorics.util](#)), 217
[_padic_lift\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 773
[_primitive_in_x0\(\)](#) (in module [diofant.polys.modulargcd](#)), 779
[_print\(\)](#) ([diofant.printing.printer.Printer](#) method), 550
[_rational_function_reconstruction\(\)](#) (in module [diofant.polys.modulargcd](#)), 779
[_rational_reconstruction_func_coeffs\(\)](#) (in module [diofant.polys.modulargcd](#)), 779
[_rational_reconstruction_int_coeffs\(\)](#) (in module [diofant.polys.modulargcd](#)), 780
[_remove_gens\(\)](#) (in module [diofant.combinatorics.util](#)), 218
[_sqf_p\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 774
[_strip\(\)](#) (in module [diofant.combinatorics.util](#)), 219
[_strong_gens_from_distr\(\)](#) (in module [diofant.combinatorics.util](#)), 220
[_subs_ground\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 774
[_test_evaluation_points\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 774
[_test_prime\(\)](#) (in module [diofant.polys.factorization_alg_field](#)), 774
[_to_ANP_poly\(\)](#) (in module [diofant.polys.modulargcd](#)), 780
[_to_ZZ_poly\(\)](#) (in module [dio-](#)

`fant.polys.modulargcd`), 780
`_trunc()` (in module `diofant.polys.modulargcd`), 781
`_undetermined_coefficients_match()` (in module `diofant.solvers.ode`), 683
`_union_find_merge()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 172
`_union_find_rep()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 173
`_verify_bsgs()` (in module `diofant.combinatorics.testutil`), 222
`_verify_centralizer()` (in module `diofant.combinatorics.testutil`), 222
`_verify_normal_closure()` (in module `diofant.combinatorics.testutil`), 222
`_z_to_alpha()` (in module `diofant.polys.factorization_alg_field`), 774
`_zz_heu_gcd()` (`diofant.polys.euclidtools._GCD` method), 783
-V
 python--m-diofant command line option, 39
--auto-symbols
 python--m-diofant command line option, 39
--help
 python--m-diofant command line option, 39
--no-ipython
 python--m-diofant command line option, 39
--no-wrap-division
 python--m-diofant command line option, 39
--unicode-identifiers
 python--m-diofant command line option, 39
--version
 python--m-diofant command line option, 39
--wrap-floats
 python--m-diofant command line option, 39
-a
 python--m-diofant command line option, 39
-h
 python--m-diofant command line option, 39

A
a (`diofant.polys.rootisolation.RealInterval` property), 784
a2idx() (in module `diofant.matrices.matrices`), 479
AbelianGroup() (in module `diofant.combinatorics.named_groups`), 214
above() (`diofant.printing.stringpict.stringPict` method), 563
Abs (class in `diofant.functions.elementary.complexes`), 276
accepted_latex_functions (in module `diofant.printing.latex`), 558
acos (class in `diofant.functions.elementary.trigonometric`), 286
acosh (class in `diofant.functions.elementary.hyperbolic`), 293
acot (class in `diofant.functions.elementary.trigonometric`), 288
acoth (class in `diofant.functions.elementary.hyperbolic`), 294
acsc (class in `diofant.functions.elementary.trigonometric`), 288
Add (class in `diofant.core.add`), 104
add() (`diofant.matrices.matrices.MatrixBase` method), 444
add() (`diofant.matrices.sparse.SparseMatrixBase` method), 490
add_formulae() (in module `diofant.simplify.hyperexpand`), 798
adjoint (class in `diofant.functions.elementary.complexes`), 277
adjoint() (`diofant.core.expr.Expr` method), 57
adjoint() (`diofant.matrices.immutable.ImmutableMatrix` method), 498
adjoint() (`diofant.matrices.matrices.MatrixBase` method), 444
adjugate() (`diofant.matrices.matrices.MatrixBase` method), 444
airyai (class in `diofant.functions.special.bessel`), 356
airyaiprime (class in `diofant.functions.special.bessel`), 359
AiryBase (class in `diofant.functions.special.bessel`), 356

airybi (class in diofant.functions.special.bessel), 357
 airybiprime (class in diofant.functions.special.bessel), 360
 algebraic_field() (diofant.domains.AlgebraicField method), 429
 algebraic_field() (diofant.domains.RationalField method), 429
 AlgebraicField (class in diofant.domains), 429
 all_coeffs() (diofant.polys.polytools.Poly method), 508
 all_roots() (diofant.polys.polytools.Poly method), 508
 all_roots() (diofant.polys.rootoftools.RootOf class method), 541
 allhints (in module diofant.solvers.ode), 643
 almosteq() (diofant.domains.ComplexField method), 431
 almosteq() (diofant.domains.RealField method), 431
 alternating() (diofant.combinatorics.generators method), 169
 AlternatingGroup() (in module diofant.combinatorics.named_groups), 214
 an (diofant.functions.special.hyper.meijerg property), 371
 And (class in diofant.logic.boolalg), 419
 annotated() (in module diofant.printing.pretty_symbology), 563
 antdivisor_count() (in module diofant.ntheory.factor_), 235
 antdivisors() (in module diofant.ntheory.factor_), 235
 aother (diofant.functions.special.hyper.meijerg property), 371
 ap (diofant.functions.special.hyper.hyper property), 369
 ap (diofant.functions.special.hyper.meijerg property), 371
 apart() (diofant.core.expr.Expr method), 57
 apart() (in module diofant.polys.partfrac), 545
 apart_list() (in module diofant.polys.partfrac), 546
 apply() (diofant.simplify.epathtools.EPath method), 604
 applyfunc() (diofant.matrices.dense.DenseMatrix method), 480
 applyfunc() (diofant.matrices.sparse.SparseMatrixBase method), 491
 approximation_interval() (diofant.core.numbers.NumberSymbol method), 91
 arg (class in diofant.functions.elementary.complexes), 277
 args (diofant.core.basic.Basic property), 46
 args (diofant.core.containers.Dict property), 140
 args (diofant.polys.polytools.Poly property), 508
 args (diofant.tensor.indexed.IndexedBase property), 707
 args_cnc() (diofant.core.expr.Expr method), 57
 Argument (class in diofant.utilities.codegen), 717
 argument (diofant.functions.special.bessel.BesselBase property), 351
 argument (diofant.functions.special.hyper.hyper property), 369
 argument (diofant.functions.special.hyper.meijerg property), 371
 Array (class in diofant.tensor.array), 700
 array_form (diofant.combinatorics.permutations.Permutation property), 151
 array_form (diofant.combinatorics.polyhedron.Polyhedron property), 196
 as_base_exp() (diofant.core.expr.Expr method), 57
 as_base_exp() (diofant.core.mul.Mul method), 101
 as_base_exp() (diofant.core.power.Pow method), 100
 as_coeff_Add() (diofant.core.add.Add method), 104
 as_coeff_add() (diofant.core.add.Add method), 104
 as_coeff_Add() (diofant.core.expr.Expr method), 58
 as_coeff_add() (diofant.core.expr.Expr method), 58
 as_coeff_Add() (diofant.core.numbers.Number method), 85
 as_coeff_add() (diofant.core.numbers.Number method), 85
 as_coeff_exponent() (diofant.core.expr.Expr method), 58
 as_coeff_Mul() (diofant.core.expr.Expr

`method)`, 58
`as_coeff_mul()` (`diofant.core.expr.Expr` `method`), 58
`as_coeff_Mul()` (`diofant.core.mul.Mul` `method`), 101
`as_coeff_mul()` (`diofant.core.mul.Mul` `method`), 101
`as_coeff_Mul()` (`diofant.core.numbers.Number` `method`), 85
`as_coeff_mul()` (`diofant.core.numbers.Number` `method`), 85
`as_coefficient()` (`diofant.core.expr.Expr` `method`), 59
`as_coefficients_dict()` (`diofant.core.add.Add` `method`), 104
`as_coefficients_dict()` (`diofant.core.expr.Expr` `method`), 60
`as_content_primitive()` (`diofant.core.add.Add` `method`), 104
`as_content_primitive()` (`diofant.core.expr.Expr` `method`), 60
`as_content_primitive()` (`diofant.core.mul.Mul` `method`), 102
`as_content_primitive()` (`diofant.core.numbers.Rational` `method`), 89
`as_content_primitive()` (`diofant.core.power.Pow` `method`), 100
`as_dict()` (`diofant.combinatorics.partitions.IntegerPartition` `method`), 144
`as_dict()` (`diofant.polys.polytools.Poly` `method`), 509
`as_dummy()` (`diofant.concrete.expr_with_limits.ExprWithLimits` `method`), 267
`as_explicit()` (`diofant.matrices.expressions.MatrixExpr` `method`), 500
`as_expr()` (`diofant.core.expr.Expr` `method`), 61
`as_expr()` (`diofant.polys.monomials.Monomial` `method`), 540
`as_expr()` (`diofant.polys.polytools.Poly` `method`), 509
`as_ferrers()` (`diofant.combinatorics.partitions.IntegerPartition` `method`), 145
`as_immutable()` (`diofant.matrices.dense.DenseMatrix` `method`), 480
`as_immutable()` (`diofant.matrices.sparse.SparseMatrixBase` `method`), 491
`as_independent()` (`diofant.core.expr.Expr` `method`), 61
`as_int()` (in module `diofant.core.compatibility`), 140
`as_leading_term()` (`diofant.core.expr.Expr` `method`), 63
`as_mutable()` (`diofant.matrices.dense.DenseMatrix` `method`), 480
`as_mutable()` (`diofant.matrices.expressions.MatrixExpr` `method`), 501
`as_mutable()` (`diofant.matrices.immutable.ImmutableMatrix` `method`), 498
`as_mutable()` (`diofant.matrices.sparse.SparseMatrixBase` `method`), 491
`as_numer_denom()` (`diofant.core.expr.Expr` `method`), 63
`as_ordered_factors()` (`diofant.core.expr.Expr` `method`), 63
`as_ordered_factors()` (`diofant.core.mul.Mul` `method`), 102
`as_ordered_terms()` (`diofant.core.expr.Expr` `method`), 63
`as_poly()` (`diofant.core.expr.Expr` `method`), 64
`as_powers_dict()` (`diofant.core.expr.Expr` `method`), 64
`as_powers_dict()` (`diofant.core.mul.Mul` `method`), 102
`as_property()` (in module `diofant.core.assumptions`), 44
`as_real_imag()` (`diofant.core.add.Add` `method`), 105
`as_real_imag()` (`diofant.core.expr.Expr` `method`), 64
`as_real_imag()` (`diofant.core.mul.Mul` `method`), 102
`as_real_imag()` (`diofant.core.power.Pow` `method`), 101
`as_real_imag()` (`diofant.functions.elementary.complexes.im` `method`), 276
`as_real_imag()` (`diofant.functions.elementary.complexes.re` `method`), 275
`as_real_imag()` (`diofant.functions.elementary.exponential.log` `method`), 296
`as_real_imag()` (`diofant.functions.elementary.hyperbolic.sinh` `method`), 291

`as_relational()` (*diofant.sets.sets.FiniteSet* method), 575
`as_relational()` (*diofant.sets.sets.Intersection* method), 576
`as_relational()` (*diofant.sets.sets.Interval* method), 573
`as_relational()` (*diofant.sets.sets.Union* method), 576
`as_set()` (*diofant.core.relational.Relational* method), 108
`as_sum()` (*diofant.integrals.integrals.Integral* method), 405
`as_terms()` (*diofant.core.expr.Expr* method), 64
`as_tuple()` (*diofant.polys.rootisolation.ComplexInterval* method), 784
`as_tuple()` (*diofant.polys.rootisolation.RealInterval* method), 784
`as_two_terms()` (*diofant.core.add.Add* method), 105
`as_two_terms()` (*diofant.core.mul.Mul* method), 102
`ascents()` (*diofant.combinatorics.permutations.Permutation* method), 152
`asec` (class in *diofant.functions.elementary.trigonometric*), 286
`aseries()` (*diofant.core.expr.Expr* method), 64
`asin` (class in *diofant.functions.elementary.trigonometric*), 285
`asinh` (class in *diofant.functions.elementary.hyperbolic*), 293
`assemble_partfrac_list()` (in module *diofant.polys.partfrac*), 547
`AssignmentError`, 562
`assoc_laguerre` (class in *diofant.functions.special.polynomials*), 384
`assoc_legendre` (class in *diofant.functions.special.polynomials*), 381
`atan` (class in *diofant.functions.elementary.trigonometric*), 287
`atan2` (class in *diofant.functions.elementary.trigonometric*), 289
`atanh` (class in *diofant.functions.elementary.hyperbolic*), 293
`Atom` (class in *diofant.core.basic*), 46
`AtomicExpr` (class in *diofant.core.expr*), 80
`atoms()` (*diofant.combinatorics.permutations.Permutation* method), 152
`atoms()` (*diofant.core.basic.Basic* method), 46
`atoms()` (*diofant.matrices.matrices.MatrixBase* method), 444
`auto_number()` (in module *diofant.parsing.sympy_parser*), 751
`auto_symbol()` (in module *diofant.parsing.sympy_parser*), 751
`AutomaticSymbols` (class in *diofant.interactive.session*), 566
`autowrap()` (in module *diofant.utilities.autowrap*), 713
`ax` (*diofant.polys.rootisolation.ComplexInterval* property), 784
`ay` (*diofant.polys.rootisolation.ComplexInterval* property), 784

B

`b` (*diofant.polys.rootisolation.RealInterval* property), 784
`base` (*diofant.combinatorics.perm_groups.PermutationGroup* property), 173
`base` (*diofant.core.power.Pow* property), 101
`base` (*diofant.tensor.indexed.Indexed* property), 705
`base_solution_linear()` (in module *diofant.solvers.diophantine*), 618
`BasePolynomialError`, 785
`baseswap()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 174
`Basic` (class in *diofant.core.basic*), 46
`basic_orbits` (*diofant.combinatorics.perm_groups.PermutationGroup* property), 175
`basic_stabilizers` (*diofant.combinatorics.perm_groups.PermutationGroup* property), 175
`basic_transversals` (*diofant.combinatorics.perm_groups.PermutationGroup* property), 175
`bell` (class in *diofant.functions.combinatorial.numbers*), 304
`below()` (*diofant.printing.stringpict.stringPict* method), 563
`berkowitz()` (*diofant.matrices.matrices.MatrixBase* method), 445
`berkowitz_charpoly()` (*diofant.matrices.matrices.MatrixBase* method), 445

berkowitz_det() (diofant.matrices.matrices.MatrixBase method), 446
berkowitz_eigenvals() (diofant.matrices.matrices.MatrixBase method), 446
berkowitz_minors() (diofant.matrices.matrices.MatrixBase method), 446
bernoulli (class in diofant.functions.combinatorial.numbers), 305
BesselBase (class in diofant.functions.special.bessel), 351
besseli (class in diofant.functions.special.bessel), 352
besselj (class in diofant.functions.special.bessel), 351
bessellk (class in diofant.functions.special.bessel), 353
besselsimp() (in module diofant.simplify.simplify), 584
bessely (class in diofant.functions.special.bessel), 352
beta (class in diofant.functions.special.beta_functions), 328
bin_to_gray() (diofant.combinatorics.graycode method), 211
binary_function() (in module diofant.utilities.autowrap), 714
binomial (class in diofant.functions.combinatorial.factorials), 306
binomial_coefficients() (in module diofant.ntheory.multinomial), 249
binomial_coefficients_list() (in module diofant.ntheory.multinomial), 249
bitlist_from_subset() (diofant.combinatorics.subsets.Subset class method), 202
block_collapse() (in module diofant.matrices.expressions.blockmatrix), 504
BlockDiagMatrix (class in diofant.matrices.expressions.blockmatrix), 504
BlockMatrix (class in diofant.matrices.expressions.blockmatrix), 503
bm (diofant.functions.special.hyper.meijerg property), 371
BooleanFalse (class in diofant.logic.boolalg), 418
BooleanTrue (class in diofant.logic.boolalg), 417
bother (diofant.functions.special.hyper.meijerg property), 371
boundary (diofant.sets.sets.Set property), 567
bq (diofant.functions.special.hyper.hyper property), 369
bq (diofant.functions.special.hyper.meijerg property), 371
bsgs_direct_product() (in module diofant.combinatorics.tensor_can), 228
bspline_basis() (in module diofant.functions.special.bsplines), 361
bspline_basis_set() (in module diofant.functions.special.bsplines), 362
buchberger() (in module diofant.polys.groebnertools), 767
bx (diofant.polys.rootisolation.ComplexInterval property), 784
by (diofant.polys.rootisolation.ComplexInterval property), 784
C
C (diofant.matrices.immutable.ImmutableMatrix property), 498
C (diofant.matrices.matrices.MatrixBase property), 440
cacheit() (in module diofant.core.cache), 45
cancel() (diofant.core.expr.Expr method), 65
cancel() (diofant.polys.polytools.Poly method), 509
cancel() (diofant.polys.rings.PolyElement method), 759
cancel() (in module diofant.polys.polytools), 527
canonical (diofant.core.relational.Relational property), 108
canonical_variables (diofant.core.expr.Expr property), 65
canonicalize() (in module diofant.combinatorics.tensor_can), 223
cantor_product() (in module diofant.utilities.iterables), 731
cardinality (diofant.combinatorics.permutations.Permutation property), 152
cardinality (diofant.combinatorics.subsets.Subset property), 203

casoratian() (in module diofant.matrices.dense), 476

Catalan (class in diofant.core.numbers), 98

catalan (class in diofant.functions.combinatorial.numbers), 307

ccode() (in module diofant.printing.ccode), 552

CCodeGen (class in diofant.utilities.codegen), 717

CCodePrinter (class in diofant.printing.ccode), 551

ceiling (class in diofant.functions.elementary.integers), 294

ceiling() (diofant.core.numbers.Float method), 87

center (diofant.polys.rootisolation.ComplexInterval property), 784

center (diofant.polys.rootisolation.RealIntervalCi property), 785

center() (diofant.combinatorics.perm_groups.PermutationGroup method), 176

centralizer() (diofant.combinatorics.perm_groups.PermutationGroup method), 176

change_index() (diofant.concrete.expr_with_intlimits.ExprWithIntLimits method), 268

characteristic (diofant.domains.ring.CommutativeRing property), 428

CharacteristicZero (class in diofant.domains.characteristiczero), 429

charpoly() (diofant.matrices.matrices.MatrixBase method), 446

chebyshevt (class in diofant.functions.special.polynomials), 378

chebyshevt_poly() (in module diofant.polys.orthopolys), 543

chebyshevt_root (class in diofant.functions.special.polynomials), 380

chebyshevu (class in diofant.functions.special.polynomials), 379

chebyshevu_poly() (in module diofant.polys.orthopolys), 543

chebyshevu_root (class in diofant.functions.special.polynomials), 380

check_assumptions() (in module diofant.core.assumptions), 44

checkinfsol() (in module diofant.solvers.ode), 642

checkodesol() (in module diofant.solvers.ode), 640

checkpdesol() (in module diofant.solvers.pde), 691

checksol() (in module diofant.solvers.utils), 695

Chi (class in diofant.functions.special.error_functions), 349

cholesky() (diofant.matrices.matrices.MatrixBase method), 447

cholesky() (diofant.matrices.sparse.SparseMatrixBase method), 491

cholesky_solve() (diofant.matrices.matrices.MatrixBase method), 447

Cholesky (class in diofant.functions.special.error_functions), 349

CL (diofant.matrices.sparse.SparseMatrixBase property), 490

class_key() (diofant.core.add.Add class method), 105

class_key() (diofant.core.basic.Atom class method), 46

class_key() (diofant.core.basic.Basic class method), 47

class_key() (diofant.core.function.Function class method), 127

class_key() (diofant.core.mul.Mul class method), 102

class_key() (diofant.core.numbers.Number class method), 85

class_key() (diofant.core.power.Pow class method), 101

class_key() (diofant.core.symbol.Dummy class method), 82

classify_diop() (in module diofant.solvers.diophantine), 616

classify_ode() (in module diofant.solvers.ode), 638

classify_pde() (in module diofant.solvers.pde), 691

classof() (in module diofant.matrices.matrices), 473

clear_denoms() (diofant.polys.polytools.Poly method), 509

closure (diofant.sets.sets.Set property), 567

CodeGen (class in diofant.utilities.codegen), 718

codegen() (in module dio-

`fant.utilities.codegen`), 721
 CodePrinter (class in `diofant.printing.codeprinter`), 562
 CodeWrapError, 712
 CodeWrapper (class in `diofant.utilities.autowrap`), 712
`coeff()` (`diofant.core.expr.Expr` method), 65
`coeff_monomial()` (`diofant.polys.polytools.Poly` method), 509
`coeffs()` (`diofant.polys.polytools.Poly` method), 510
 CoercionFailedError, 785
`cofactor()` (`diofant.matrices.matrices.MatrixBase` method), 447
`cofactorMatrix()` (`diofant.matrices.matrices.MatrixBase` method), 448
`cofactors()` (`diofant.core.numbers.Number` method), 85
`cofactors()` (`diofant.domains.ring.CommutativeRing` method), 428
`cofactors()` (`diofant.polys.polytools.Poly` method), 510
`cofactors()` (in module `diofant.polys.polytools`), 527
`col_insert()` (`diofant.matrices.matrices.MatrixBase` method), 448
`col_join()` (`diofant.matrices.matrices.MatrixBase` method), 448
`col_join()` (`diofant.matrices.sparse.MutableSparseMatrix` method), 487
`col_list()` (`diofant.matrices.sparse.SparseMatrixBase` method), 492
`col_op()` (`diofant.matrices.dense.MutableDenseMatrix` method), 482
`col_op()` (`diofant.matrices.sparse.MutableSparseMatrix` method), 487
`col_swap()` (`diofant.matrices.dense.MutableDenseMatrix` method), 482
`col_swap()` (`diofant.matrices.sparse.MutableSparseMatrix` method), 488
`collect()` (`diofant.core.expr.Expr` method), 67
`collect()` (in module `diofant.simplify.radsimp`), 588
`collect_const()` (in module `diofant.simplify.radsimp`), 592
`collect_sqrt()` (in module `diofant.simplify.radsimp`), 591
`combsimp()` (`diofant.core.expr.Expr` method), 67
`combsimp()` (in module `diofant.simplify.combsimp`), 599
`common_prefix()` (in module `diofant.utilities.iterables`), 731
`common_suffix()` (in module `diofant.utilities.iterables`), 731
 CommutativeRing (class in `diofant.domains.ring`), 428
`commutator()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 177
`commutator()` (`diofant.combinatorics.permutations.Permutation` method), 152
`commutes_with()` (`diofant.combinatorics.permutations.Permutation` method), 153
 Complement (class in `diofant.sets.sets`), 577
`complement()` (`diofant.sets.sets.Set` method), 568
 ComplexAlgebraicField (class in `diofant.domains`), 429
 ComplexField (class in `diofant.domains`), 431
 ComplexInfinity (class in `diofant.core.numbers`), 95
 ComplexInterval (class in `diofant.polys.rootisolation`), 784
`components()` (in module `diofant.integrals.heurisch`), 403
`compose()` (`diofant.polys.fields.FracElement` method), 786
`compose()` (`diofant.polys.polytools.Poly` method), 510
`compose()` (`diofant.polys.rings.PolyElement` method), 759
`compose()` (in module `diofant.polys.polytools`), 528
 CompositeDomain (class in `diofant.domains.compositedomain`), 428
 CoercionFailedError, 785
`compute_leading_term()` (`diofant.core.expr.Expr` method), 67
`cond` (`diofant.functions.elementary.piecewise.ExprCondPair` property), 297
`condition_number()` (`diofant.matrices.matrices.MatrixBase` method), 448
 conjugate (class in `diofant.functions.elementary.complexes`), 278
`conjugate` (`diofant.combinatorics.partitions.IntegerPartition` property), 145
`conjugate()` (`diofant.core.expr.Expr` method), 67

`conjugate()` (`diofant.matrices.immutable.ImmutableMatrix` method), 498
`conjugate()` (`diofant.matrices.matrices.MatrixBase` method), 449
`conjugate()` (`diofant.polys.rootisolation.ComplexInterval` method), 784
`conserve_mpmath_dps()` (in module `diofant.utilities.decorator`), 725
`constant_renumber()` (in module `diofant.solvers.ode`), 644
`constantsimp()` (in module `diofant.solvers.ode`), 645
`construct_domain()` (in module `diofant.polys.constructor`), 539
`contains()` (`diofant.calculus.order.Order` method), 755
`contains()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 177
`contains()` (`diofant.polys.polytools.GroebnerBasis` method), 505
`contains()` (`diofant.sets.sets.Set` method), 568
`content()` (`diofant.polys.polytools.Poly` method), 511
`content()` (`diofant.polys.rings.PolyElement` method), 759
`content()` (in module `diofant.polys.polytools`), 528
`continued_fraction_convergents()` (in module `diofant.ntheory.continued_fraction`), 257
`continued_fraction_iterator()` (in module `diofant.ntheory.continued_fraction`), 257
`continued_fraction_periodic()` (in module `diofant.ntheory.continued_fraction`), 258
`continued_fraction_reduce()` (in module `diofant.ntheory.continued_fraction`), 259
`convergence_statement` (`diofant.functions.special.hyper.hyper` property), 369
`convert()` (`diofant.domains.domain.Domain` method), 427
`convert_from()` (`diofant.domains.domain.Domain` method), 427
`convert_xor()` (in module `diofant.parsing.sympy_parser`), 751
`copy()` (`diofant.core.basic.Basic` method), 47
`copy()` (`diofant.matrices.matrices.MatrixBase` method), 449
`copy()` (`diofant.polys.rings.PolyElement` method), 759
`copyin_list()` (`diofant.matrices.dense.MutableDenseMatrix` method), 482
`copyin_matrix()` (`diofant.matrices.dense.MutableDenseMatrix` method), 483
`core()` (in module `diofant.ntheory.factor_`), 236
`cornacchia()` (in module `diofant.solvers.diophantine`), 620
`corners` (`diofant.combinatorics.polyhedron.Polyhedron` property), 196
`cos` (class in `diofant.functions.elementary.trigonometric`), 283
`coset_factor()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 178
`coset_rank()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 179
`coset_unrank()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 179
`cosh` (class in `diofant.functions.elementary.hyperbolic`), 291
`cosine_transform()` (in module `diofant.integrals.transforms`), 397
`CosineTransform` (class in `diofant.integrals.transforms`), 410
`cot` (class in `diofant.functions.elementary.trigonometric`), 283
`coth` (class in `diofant.functions.elementary.hyperbolic`), 292
`could_extract_minus_sign()` (`diofant.core.expr.Expr` method), 67
`count()` (`diofant.core.basic.Basic` method), 47
`count_ops()` (`diofant.core.basic.Basic` method), 47
`count_ops()` (`diofant.core.expr.Expr` method), 67
`count_ops()` (in module `diofant.core.function`), 133
`count_partitions()` (`diofant.utilities.enumerative.MultisetPartitionTraverse` method), 727
`count_roots()` (`diofant.polys.polytools.Poly` method), 511

`count_roots()` (in module `diofant.polys.polytools`), 528
`cp_key()` (in module `diofant.polys.groebnertools`), 768
`critical_pair()` (in module `diofant.polys.groebnertools`), 768
`cross()` (`diofant.matrices.matrices.MatrixBasedefault_sort_key()` (in module `diofant.utilities.iterables`), 731
`crt()` (in module `diofant.ntheory.modular`), 246
`crt1()` (in module `diofant.ntheory.modular`), 247
`crt2()` (in module `diofant.ntheory.modular`), 247
`csc` (class in `diofant.functions.elementary.trigonometric`), 284
`csch` (class in `diofant.functions.elementary.hyperbolic`), 292
`cse()` (in module `diofant.simplify.cse_main`), 601
`current` (`diofant.combinatorics.graycode.GrayCode` property), 209
`Cycle` (class in `diofant.combinatorics.permutations`), 167
`cycle_length()` (in module `diofant.ntheory.generate`), 230
`cycle_structure` (`diofant.combinatorics.permutations.Permutation` property), 153
`cycles` (`diofant.combinatorics.permutations.Permutation` property), 153
`cyclic()` (`diofant.combinatorics.generators` method), 169
`cyclic_form` (`diofant.combinatorics.permutations.Permutation` property), 154
`cyclic_form` (`diofant.combinatorics.polyhedron.Polyhedron` property), 197
`CyclicGroup()` (in module `diofant.combinatorics.named_groups`), 213
`cyclotomic_poly()` (in module `diofant.polys.specialpolys`), 543
`CythonCodeWrapper` (class in `diofant.utilities.autowrap`), 712
D
`D` (`diofant.matrices.matrices.MatrixBase` property), 440
`DataType` (class in `diofant.utilities.codegen`), 719
`decompose()` (`diofant.polys.polytools.Poly` method), 511
`decompose()` (`diofant.polys.univar.UnivarPolyElement` method), 763
`decompose()` (in module `diofant.polys.polytools`), 528
`default_sort_key()` (in module `diofant.utilities.iterables`), 731
`degree` (`diofant.combinatorics.perm_groups.PermutationGroup` property), 179
`degree()` (`diofant.polys.polytools.Poly` method), 511
`degree()` (`diofant.polys.rings.PolyElement` method), 760
`degree()` (in module `diofant.polys.polytools`), 528
`delta` (`diofant.functions.special.hyper.meijerg` property), 371
`deltaintegrate()` (in module `diofant.integrals.deltafunctions`), 401
`DenseMatrix` (class in `diofant.matrices.dense`), 479
`Derivative` (class in `diofant.core.function`), 122
`derive_by_array()` (in module `diofant.tensor.array`), 700
`derived_series()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 180
`derived_subgroup()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 180
`descent()` (in module `diofant.solvers.diophantine`), 624
`descents()` (`diofant.combinatorics.permutations.Permutation` method), 154
`det` (`diofant.matrices.matrices.MatrixBase` method), 449
`det` (`diofant.matrices.matrices.MatrixBase` method), 449
`det_LU_decomposition()` (`diofant.matrices.matrices.MatrixBase` method), 449
`diag()` (in module `diofant.matrices.dense`), 474
`diagonal_solve()` (`diofant.matrices.matrices.MatrixBase` method), 450
`diagonalize()` (`diofant.matrices.matrices.MatrixBase` method), 450
`Dict` (class in `diofant.core.containers`), 139
`diff()` (`diofant.core.expr.Expr` method), 67

[diff\(\)](#) ([diofant.matrices.immutable.ImmutableMatrix](#) module, 202
[method](#)), 498
[diff\(\)](#) ([diofant.matrices.matrices.MatrixBase](#)
[method](#)), 450
[diff\(\)](#) ([diofant.polys.fields.FracElement](#)
[method](#)), 786
[diff\(\)](#) ([diofant.polys.rings.PolyElement](#)
[method](#)), 760
[diff\(\)](#) (in module [diofant.core.function](#)), 125
[digamma\(\)](#) (in module [diofant.functions.special.gamma_functions](#)), 324
[dihedral\(\)](#) ([diofant.combinatorics.generators](#)
[method](#)), 169
[DihedralGroup\(\)](#) (in module [diofant.combinatorics.named_groups](#)), 213
[dimension](#) ([diofant.polys.polytools.GroebnerBasis](#)
[property](#)), 505
[diofant](#)
[module](#), 41
[diofant.calculus](#)
[module](#), 752
[diofant.calculus.gruntz](#)
[module](#), 787
[diofant.calculus.limits](#)
[module](#), 752
[diofant.calculus.optimization](#)
[module](#), 753
[diofant.calculus.order](#)
[module](#), 753
[diofant.calculus.residues](#)
[module](#), 755
[diofant.calculus.singularities](#)
[module](#), 752
[diofant.combinatorics.generators](#)
[module](#), 169
[diofant.combinatorics.graycode](#)
[module](#), 208
[diofant.combinatorics.group_constructs](#)
[module](#), 221
[diofant.combinatorics.named_groups](#)
[module](#), 212
[diofant.combinatorics.partitions](#)
[module](#), 142
[diofant.combinatorics.perm_groups](#)
[module](#), 170
[diofant.combinatorics.permutations](#)
[module](#), 147
[diofant.combinatorics.polyhedron](#)
[module](#), 196
[diofant.combinatorics.prufer](#)
[module](#), 199
[diofant.combinatorics.subsets](#)
[diofant.combinatorics.tensor_can](#)
[module](#), 223
[diofant.combinatorics.testutil](#)
[module](#), 221
[diofant.combinatorics.util](#)
[module](#), 215
[diofant.config](#)
[module](#), 41
[diofant.core](#)
[module](#), 41
[diofant.core.add](#)
[module](#), 104
[diofant.core.assumptions](#)
[module](#), 44
[diofant.core.basic](#)
[module](#), 46
[diofant.core.cache](#)
[module](#), 45
[diofant.core.compatibility](#)
[module](#), 140
[diofant.core.containers](#)
[module](#), 139
[diofant.core.core](#)
[module](#), 55
[diofant.core.evalf](#)
[module](#), 138
[diofant.core.evaluate](#)
[module](#), 56
[diofant.core.expr](#)
[module](#), 57
[diofant.core.exprtools](#)
[module](#), 141
[diofant.core.function](#)
[module](#), 121
[diofant.core.mod](#)
[module](#), 106
[diofant.core.mul](#)
[module](#), 101
[diofant.core.multidimensional](#)
[module](#), 120
[diofant.core.numbers](#)
[module](#), 85
[diofant.core.power](#)
[module](#), 99
[diofant.core.relational](#)
[module](#), 107
[diofant.core.singleton](#)
[module](#), 55
[diofant.core.symbol](#)
[module](#), 80
[diofant.core.sympify](#)
[module](#), 41
[diofant.domains](#)

- module, [426](#)
- diofant.functions
 - module, [274](#)
- diofant.functions.special.bessel
 - module, [351](#)
- diofant.functions.special.beta_functions
 - module, [328](#)
- diofant.functions.special.elliptic_integrals
 - module, [372](#)
- diofant.functions.special.error_functions
 - module, [329](#)
- diofant.functions.special.gamma_functions
 - module, [319](#)
- diofant.functions.special.polynomials
 - module, [374](#)
- diofant.functions.special.zeta_functions
 - module, [363](#)
- diofant.integrals
 - module, [392](#)
- diofant.integrals.meijerint_doc
 - module, [809](#)
- diofant.integrals.quadrature
 - module, [411](#)
- diofant.integrals.transforms
 - module, [393](#)
- diofant.interactive
 - module, [566](#)
- diofant.interactive.printing
 - module, [566](#)
- diofant.interactive.session
 - module, [566](#)
- diofant.logic
 - module, [417](#)
- diofant.matrices
 - module, [432](#)
- diofant.matrices.dense
 - module, [479](#)
- diofant.matrices.expressions
 - module, [500](#)
- diofant.matrices.expressions.blockmatrix
 - module, [503](#)
- diofant.matrices.immutable
 - module, [498](#)
- diofant.matrices.matrices
 - module, [432](#)
- diofant.matrices.sparse
 - module, [487](#)
- diofant.ntheory.continued_fraction
 - module, [257](#)
- diofant.ntheory.egyptian_fraction
 - module, [259](#)
- diofant.ntheory.factor_
 - module, [235](#)
- diofant.ntheory.generate
 - module, [228](#)
- diofant.ntheory.modular
 - module, [246](#)
- diofant.ntheory.multinomial
 - module, [249](#)
- diofant.ntheory.partitions_
 - module, [250](#)
- diofant.ntheory.primetest
 - module, [250](#)
- diofant.ntheory.residue_ntheory
 - module, [252](#)
- diofant.parsing
 - module, [748](#)
- diofant.polys
 - module, [505](#)
- diofant.polys.constructor
 - module, [539](#)
- diofant.polys.euclidtools
 - module, [764](#)
- diofant.polys.factorization_alg_field
 - module, [771](#)
- diofant.polys.factortools
 - module, [767](#)
- diofant.polys.groebnertools
 - module, [767](#)
- diofant.polys.modulargcd
 - module, [775](#)
- diofant.polys.monomials
 - module, [540](#)
- diofant.polys.numberfields
 - module, [539](#)
- diofant.polys.orderings
 - module, [540](#)
- diofant.polys.orthopolys
 - module, [543](#)
- diofant.polys.partfrac
 - module, [545](#)
- diofant.polys.polyerrors
 - module, [785](#)
- diofant.polys.polyfuncs
 - module, [537](#)
- diofant.polys.polyoptions
 - module, [785](#)
- diofant.polys.polyroots
 - module, [542](#)
- diofant.polys.polytools
 - module, [505](#)
- diofant.polys.rationaltools
 - module, [544](#)
- diofant.polys.rootisolation
 - module, [784](#)
- diofant.polys.rootoftools
 - module, [540](#)
- diofant.polys.specialpolys
 - module, [228](#)

- module, [543](#)
- diofant.polys.sqfreetools
 - module, [785](#)
- diofant.printing
 - module, [548](#)
- diofant.printing.ccode
 - module, [551](#)
- diofant.printing.codeprinter
 - module, [562](#)
- diofant.printing.conventions
 - module, [562](#)
- diofant.printing.fcode
 - module, [554](#)
- diofant.printing.lambdarepr
 - module, [558](#)
- diofant.printing.latex
 - module, [558](#)
- diofant.printing.mathematica
 - module, [557](#)
- diofant.printing.mathml
 - module, [560](#)
- diofant.printing.precedence
 - module, [562](#)
- diofant.printing.pretty
 - module, [551](#)
- diofant.printing.pretty_symbology
 - module, [562](#)
- diofant.printing.printer
 - module, [548](#)
- diofant.printing.python
 - module, [561](#)
- diofant.printing.repr
 - module, [561](#)
- diofant.printing.str
 - module, [561](#)
- diofant.printing.stringpict
 - module, [563](#)
- diofant.sets.fancysets
 - module, [578](#)
- diofant.sets.sets
 - module, [567](#)
- diofant.simplify.combsimp
 - module, [599](#)
- diofant.simplify.cse_main
 - module, [601](#)
- diofant.simplify.epathtools
 - module, [604](#)
- diofant.simplify.fu
 - module, [595](#)
- diofant.simplify.hyperexpand
 - module, [603](#)
- diofant.simplify.hyperexpand_doc
 - module, [797](#)
- diofant.simplify.powsimp
 - module, [596](#)
- diofant.simplify.radsimp
 - module, [587](#)
- diofant.simplify.ratsimp
 - module, [593](#)
- diofant.simplify.sqrtdenest
 - module, [599](#)
- diofant.simplify.traversaltools
 - module, [604](#)
- diofant.simplify.trigsimp
 - module, [593](#)
- diofant.solvers
 - module, [606](#)
- diofant.solvers.deutils
 - module, [696](#)
- diofant.solvers.diophantine
 - module, [610](#)
- diofant.solvers.inequalities
 - module, [610](#)
- diofant.solvers.ode
 - module, [680](#)
- diofant.solvers.pde
 - module, [695](#)
- diofant.solvers.polysys
 - module, [608](#)
- diofant.solvers.recurr
 - module, [684](#)
- diofant.solvers.solvers
 - module, [606](#)
- diofant.solvers.utils
 - module, [695](#)
- diofant.tensor
 - module, [696](#)
- diofant.tensor.array
 - module, [697](#)
- diofant.tensor.index_methods
 - module, [708](#)
- diofant.tensor.indexed
 - module, [702](#)
- diofant.utilities
 - module, [710](#)
- diofant.utilities.autowrap
 - module, [711](#)
- diofant.utilities.codegen
 - module, [716](#)
- diofant.utilities.decorator
 - module, [725](#)
- diofant.utilities.enumerative
 - module, [725](#)
- diofant.utilities.iterables
 - module, [731](#)
- diofant.utilities.lambdify
 - module, [744](#)
- diofant.utilities.memoization

module, 747
 diofant.utilities.misc
 module, 747
 diofant.utilities.randtest
 module, 747
 diop_bf_DN() (in module diofant.solvers.diophantine), 621
 diop_DN() (in module diofant.solvers.diophantine), 619
 diop_general_pythagorean() (in module diofant.solvers.diophantine), 624
 diop_general_sum_of_even_powers() (in module diofant.solvers.diophantine), 625
 diop_general_sum_of_squares() (in module diofant.solvers.diophantine), 625
 diop_linear() (in module diofant.solvers.diophantine), 618
 diop_quadratic() (in module diofant.solvers.diophantine), 619
 diop_solve() (in module diofant.solvers.diophantine), 617
 diop_ternary_quadratic() (in module diofant.solvers.diophantine), 623
 diop_ternary_quadratic_normal() (in module diofant.solvers.diophantine), 631
 diophantine() (in module diofant.solvers.diophantine), 615
 DiracDelta (class in diofant.functions.special.delta_functions), 318
 DirectProduct() (in module diofant.combinatorics.group_constructs), 221
 dirichlet_eta (class in diofant.functions.special.zeta_functions), 364
 discrete_log() (in module diofant.ntheory.residue_ntheory), 252
 discriminant() (diofant.polys.polytools.Poly method), 511
 discriminant() (diofant.polys.rings.PolyElement method), 760
 discriminant() (in module diofant.polys.polytools), 529
 dispersionset() (diofant.polys.polytools.Poly method), 512
 dispersionset() (diofant.polys.univar.UnivarPolynomialRing method), 430
 div() (diofant.domains.field.Field method), 427
 div() (diofant.domains.ring.CommutativeRing method), 428
 div() (diofant.polys.polytools.Poly method), 512
 div() (diofant.polys.rings.PolyElement method), 760
 div() (in module diofant.polys.polytools), 529
 divides() (diofant.polys.monomials.Monomial method), 540
 divisible() (in module diofant.solvers.diophantine), 629
 divisor_count() (in module diofant.ntheory.factor_), 236
 divisor_sigma (class in diofant.ntheory.factor_), 237
 divisors() (in module diofant.ntheory.factor_), 237
 doctest_depends_on() (in module diofant.utilities.decorator), 725
 doit() (diofant.calculus.limits.Limit method), 753
 doit() (diofant.core.basic.Atom method), 46
 doit() (diofant.core.basic.Basic method), 47
 doit() (diofant.core.function.Derivative method), 124
 doit() (diofant.core.function.Subs method), 128
 doit() (diofant.functions.elementary.piecewise.Piecewise method), 298
 doit() (diofant.integrals.integrals.Integral method), 406
 doit() (diofant.integrals.transforms.IntegralTransform method), 409
 doit_numerically() (diofant.core.function.Derivative method), 125
 Domain (class in diofant.domains.domain), 427
 domain (diofant.polys.polytools.Poly property), 512
 DomainError, 785
 doprint() (diofant.printing.mathematica.MCodePrinter method), 557
 doprint() (diofant.printing.mathml.MathMLPrinter method), 560
 doprint() (diofant.printing.printer.Printer method), 550
 dot() (diofant.matrices.matrices.MatrixBase method), 451
 dotprint() (in module diofant.printing.dot), 565
 double_coset_can_rep() (in module diofant.combinatorics.tensor_can), 225

`drop()` (*diofant.polys.polytools.Poly* method), 512
`drop()` (*diofant.polys.rings.PolynomialRing* method), 429
`dsolve()` (in module *diofant.solvers.ode*), 635
`dtype` (*diofant.domains.ExpressionDomain* attribute), 431
`dual()` (*diofant.matrices.matrices.MatrixBase* method), 451
`Dummy` (class in *diofant.core.symbol*), 82
`DummyWrapper` (class in *diofant.utilities.autowrap*), 712
`dump_c()` (*diofant.utilities.autowrap.UfuncifyCodeWrapper* method), 713
`dump_c()` (*diofant.utilities.codegen.CCodeGen* method), 718
`dump_code()` (*diofant.utilities.codegen.CodeGen* method), 718
`dump_f95()` (*diofant.utilities.codegen.FCodeGen* method), 719
`dump_h()` (*diofant.utilities.codegen.CCodeGen* method), 718
`dump_h()` (*diofant.utilities.codegen.FCodeGen* method), 720
`dump_m()` (*diofant.utilities.codegen.OctaveCodeWrapper* method), 720
`dump_pyx()` (*diofant.utilities.autowrap.CythonCodeWrapper* method), 712

E

`E1()` (in module *diofant.functions.special.error_functions*), 342
`EC()` (*diofant.polys.polytools.Poly* method), 507
`edges` (*diofant.combinatorics.polyhedron.Polyhedron* property), 197
`edges()` (*diofant.combinatorics.prufer.Prufer* static method), 199
`efactor()` (in module *diofant.polys.factorization_alg_field*), 775
`egyptian_fraction()` (in module *diofant.ntheory.egyptian_fraction*), 259
`Ei` (class in *diofant.functions.special.error_functions*), 339
`eigenvals()` (*diofant.matrices.matrices.MatrixBase* method), 451
`eigenvects()` (*diofant.matrices.matrices.MatrixBase* method), 451
`Eijk()` (in module *diofant.functions.special.tensor_functions*), 388
`eject()` (*diofant.polys.polytools.Poly* method), 512
`eject()` (*diofant.polys.rings.PolynomialRing* method), 429
`elements` (*diofant.combinatorics.perm_groups.Permutation* property), 181
`eliminate()` (in module *diofant.polys.polytools*), 529
`elliptic_e` (class in *diofant.functions.special.elliptic_integrals*), 373
`elliptic_f` (class in *diofant.functions.special.elliptic_integrals*), 372
`elliptic_k` (class in *diofant.functions.special.elliptic_integrals*), 372
`elliptic_pi` (class in *diofant.functions.special.elliptic_integrals*), 373
`EM()` (*diofant.polys.polytools.Poly* method), 507
`EmptyPrinter()` (*diofant.printing.repr.ReprPrinter* method), 561
`EmptySet` (class in *diofant.sets.sets*), 578
`end` (*diofant.sets.sets.Interval* property), 573
`enum_all()` (*diofant.utilities.enumerative.MultisetPartitionTraverse* method), 728
`enum_large()` (*diofant.utilities.enumerative.MultisetPartitionTraverse* method), 729
`enum_range()` (*diofant.utilities.enumerative.MultisetPartitionTraverse* method), 729
`enum_small()` (*diofant.utilities.enumerative.MultisetPartitionTraverse* method), 730
`EPath` (class in *diofant.simplify.epathtools*), 604
`epath()` (in module *diofant.simplify.epathtools*), 605
`epsilon_eq()` (*diofant.core.numbers.Float* method), 87
`Eq` (in module *diofant.core.relational*), 107
`Equality` (class in *diofant.core.relational*), 109
`equals()` (*diofant.core.expr.Expr* method), 67
`equals()` (*diofant.core.relational.Relational* method), 108

[equals\(\)](#) (*diofant.matrices.dense.DenseMatrix* method), 541
[equals\(\)](#) (*diofant.matrices.expressions.MatrixExpr* method), 138
[equals\(\)](#) (*diofant.matrices.immutable.ImmutableMatrix* method), 128
[Equivalent](#) (class in *diofant.logic.boolalg*), 422
[equivalent\(\)](#) (in module *diofant.solvers.diophantine*), 630
[erf](#) (class in *diofant.functions.special.error_functions*), 329
[erf2](#) (class in *diofant.functions.special.error_functions*), 333
[erf2inv](#) (class in *diofant.functions.special.error_functions*), 335
[erfc](#) (class in *diofant.functions.special.error_functions*), 330
[erfcinv](#) (class in *diofant.functions.special.error_functions*), 335
[erfi](#) (class in *diofant.functions.special.error_functions*), 331
[erfinv](#) (class in *diofant.functions.special.error_functions*), 334
[ET\(\)](#) (*diofant.polys.polytools.Poly* method), 507
[eta](#) (*diofant.functions.special.hyper.hyper* property), 369
[euler](#) (class in *diofant.functions.combinatorial.numbers*), 309
[euler_maclaurin\(\)](#) (*diofant.concrete.summations.Sum* method), 263
[EulerGamma](#) (class in *diofant.core.numbers*), 97
[eval\(\)](#) (*diofant.functions.special.tensor_functions.KroneckerDelta* class method), 389
[eval\(\)](#) (*diofant.polys.polytools.Poly* method), 513
[eval_expr\(\)](#) (in module *diofant.parsing.sympy_parser*), 749
[eval_leivicivita\(\)](#) (in module *diofant.functions.special.tensor_functions*), 388
[eval_rational\(\)](#) (*diofant.polys.rootoftools.RootOf* method), 480
[evalf\(\)](#) (*diofant.core.evalf.EvalfMixin* method), 138
[evalf\(\)](#) (*diofant.core.function.Subs* method), 128
[evalf\(\)](#) (*diofant.matrices.matrices.MatrixBase* method), 452
[EvalfMixin](#) (class in *diofant.core.evalf*), 138
[evaluate\(\)](#) (in module *diofant.core.evaluate*), 56
[EvaluationFailedError](#), 785
[ExactQuotientFailedError](#), 785
[exclude\(\)](#) (*diofant.polys.polytools.Poly* method), 513
[exp](#) (*diofant.core.power.Pow* property), 101
[exp](#) (*diofant.functions.elementary.exponential.exp_polar* property), 296
[exp\(\)](#) (*diofant.matrices.matrices.MatrixBase* method), 452
[exp\(\)](#) (in module *diofant.functions.elementary.exponential*), 295
[Exp1](#) (class in *diofant.core.numbers*), 96
[exp_polar](#) (class in *diofant.functions.elementary.exponential*), 295
[expand\(\)](#) (*diofant.core.expr.Expr* method), 67
[expand\(\)](#) (*diofant.matrices.matrices.MatrixBase* method), 452
[expand\(\)](#) (in module *diofant.core.function*), 129
[expand_complex\(\)](#) (in module *diofant.core.function*), 136
[expand_func\(\)](#) (in module *diofant.core.function*), 135
[expand_log\(\)](#) (in module *diofant.core.function*), 135
[expand_mul\(\)](#) (in module *diofant.core.function*), 134
[expand_multinomial\(\)](#) (in module *diofant.core.function*), 136
[expand_power_base\(\)](#) (in module *diofant.core.function*), 137
[expand_power_exp\(\)](#) (in module *diofant.core.function*), 136
[expand_trig\(\)](#) (in module *diofant.core.function*), 135
[expint](#) (class in *diofant.functions.special.error_functions*), 341
[Expr](#) (class in *diofant.core.expr*), 57
[expr](#) (*diofant.core.function.Derivative* property), 125

- `expr` (`diofant.core.function.Lambda` property), 121
- `expr` (`diofant.core.function.Subs` property), 129
- `expr` (`diofant.functions.elementary.piecewise.ExprCondPair` property), 297
- `ExprCondPair` (class in `diofant.functions.elementary.piecewise`), 297
- `ExpressionDomain` (class in `diofant.domains`), 431
- `ExpressionDomain.Expression` (class in `diofant.domains`), 431
- `ExprWithIntLimits` (class in `diofant.concrete.expr_with_intlimits`), 268
- `ExprWithLimits` (class in `diofant.concrete.expr_with_limits`), 267
- `exquo()` (`diofant.domains.field.Field` method), 427
- `exquo()` (`diofant.domains.ring.CommutativeRing` method), 428
- `exquo()` (`diofant.polys.polytools.Poly` method), 513
- `exquo()` (in module `diofant.polys.polytools`), 529
- `exquo_ground()` (`diofant.polys.polytools.Poly` method), 514
- `extend()` (`diofant.ntheory.generate.Sieve` method), 229
- `extend_to_no()` (`diofant.ntheory.generate.Sieve` method), 229
- `ExtendedReals` (class in `diofant.sets.fancysets`), 580
- `extract()` (`diofant.matrices.matrices.MatrixBase` method), 452
- `extract()` (`diofant.matrices.sparse.SparseMatrixBase` method), 492
- `extract_additively()` (`diofant.core.expr.Expr` method), 68
- `extract_branch_factor()` (`diofant.core.expr.Expr` method), 68
- `extract_multiplicatively()` (`diofant.core.expr.Expr` method), 68
- `ExtraneousFactorsError`, 785
- `eye()` (`diofant.matrices.dense.DenseMatrix` class method), 481
- `eye()` (`diofant.matrices.sparse.SparseMatrixBase` class method), 492
- `eye()` (in module `diofant.matrices.dense`), 474
- F**
- `F2PyCodeWrapper` (class in `diofant.utilities.autowrap`), 713
- `f5_reduce()` (in module `diofant.polys.groebnertools`), 768
- `f5b()` (in module `diofant.polys.groebnertools`), 768
- `faces` (`diofant.combinatorics.polyhedron.Polyhedron` property), 197
- `factor()` (`diofant.core.expr.Expr` method), 69
- `factor()` (in module `diofant.polys.polytools`), 530
- `factor_list()` (`diofant.polys.polytools.Poly` method), 514
- `factor_list()` (in module `diofant.polys.polytools`), 530
- `factor_terms()` (in module `diofant.core.exprtools`), 141
- `factorial` (class in `diofant.functions.combinatorial.factorials`), 310
- `factorial2` (class in `diofant.functions.combinatorial.factorials`), 311
- `factoring_visitor()` (in module `diofant.utilities.enumerative`), 726
- `factorint()` (in module `diofant.ntheory.factor_`), 238
- `factorrat()` (in module `diofant.ntheory.factor_`), 240
- `factors()` (`diofant.core.numbers.Rational` method), 89
- `FallingFactorial` (class in `diofant.functions.combinatorial.factorials`), 312
- `fcode()` (in module `diofant.printing.fcode`), 554
- `fcodegen` (class in `diofant.utilities.codegen`), 719
- `FCodePrinter` (class in `diofant.printing.fcode`), 555
- `fdiff()` (`diofant.core.function.Function` method), 127
- `fdiff()` (`diofant.functions.elementary.complexes.Abs` method), 277
- `fdiff()` (`diofant.functions.elementary.exponential.Lambert` method), 296
- `fdiff()` (`diofant.functions.elementary.exponential.log` method), 297
- `fdiff()` (`diofant.functions.elementary.hyperbolic.csch` method), 293
- `fdiff()` (`diofant.functions.elementary.hyperbolic.sinh` method), 291

fibonacci (class in diofant.functions.combinatorial.numbers), 786
fraction() (in module diofant.simplify.radsimp), 592
Field (class in diofant.domains.field), 427
field (diofant.domains.field.Field property), 427
field (diofant.domains.IntegerRing property), 429
field (diofant.polys.rings.PolynomialRing property), 430
field_isomorphism() (in module diofant.polys.numberfields), 539
fill() (diofant.matrices.dense.MutableDenseMatrix method), 483
fill() (diofant.matrices.sparse.MutableSparseMatrix method), 488
filldedent() (in module diofant.utilities.misc), 747
find() (diofant.core.basic.Basic method), 48
find_DN() (in module diofant.solvers.diophantine), 623
findrecur() (diofant.concrete.summations.Sum method), 264
finite_field() (diofant.domains.IntegerRing method), 429
finite_ring() (diofant.domains.IntegerRing method), 429
FiniteField (class in diofant.domains), 429
FiniteSet (class in diofant.sets.sets), 575
FlagError, 785
flatten() (diofant.core.add.Add class method), 105
flatten() (diofant.core.mul.Mul class method), 103
flatten() (in module diofant.utilities.iterables), 733
Float (class in diofant.core.numbers), 85
FloatRationalizer (class in diofant.interactive.session), 566
floor (class in diofant.functions.elementary.integers), 294
floor() (diofant.core.numbers.Float method), 87
fourier_transform() (in module diofant.integrals.transforms), 395
FourierTransform (class in diofant.integrals.transforms), 410
frac_field() (diofant.domains.domain.Domain method), 427
FracElement (class in diofant.polys.fields), 431
free_symbols (diofant.concrete.expr_with_limits.ExprWithLimits property), 267
free_symbols (diofant.core.basic.Basic property), 48
free_symbols (diofant.core.function.Derivative property), 125
free_symbols (diofant.core.function.Lambda property), 121
free_symbols (diofant.core.function.Subs property), 129
free_symbols (diofant.integrals.integrals.Integral property), 407
free_symbols (diofant.integrals.transforms.IntegralTransform property), 409
free_symbols (diofant.matrices.matrices.MatrixBase property), 452
free_symbols (diofant.polys.polytools.Poly property), 514
free_symbols (diofant.polys.polytools.PurePoly property), 527
free_symbols_in_domain (diofant.polys.polytools.Poly property), 514
fresnelc (class in diofant.functions.special.error_functions), 338
FresnelIntegral (class in diofant.functions.special.error_functions), 336
fresnels (class in diofant.functions.special.error_functions), 336
from_dict() (diofant.polys.polytools.Poly class method), 515
from_expr() (diofant.domains.domain.Domain method), 427
from_expr() (diofant.polys.polytools.Poly class method), 515
from_inversion_vector() (diofant.combinatorics.permutations.Permutation class method), 154
from_list() (diofant.polys.polytools.Poly class method), 515

`from_poly()` (*diofant.polys.polytools.Poly* class method), 515
`from_rgs()` (*diofant.combinatorics.partitions.Partition* class method), 143
`from_sequence()` (*diofant.combinatorics.permutations.Permutation* class method), 154
`fu()` (in module *diofant.simplify.fu*), 595
`full_cyclic_form` (*diofant.combinatorics.permutations.Permutation* property), 155
`func` (*diofant.core.basic.Basic* property), 48
`func_field_modgcd()` (in module *diofant.polys.modulargcd*), 781
`Function` (class in *diofant.core.function*), 126
`function` (*diofant.concrete.expr_with_limits.ExprWithLimits* property), 267
`function` (*diofant.integrals.transforms.IntegralTransform* property), 409
`function_exponentiation()` (in module *diofant.parsing.sympy_parser*), 751
`function_variable` (*diofant.integrals.transforms.IntegralTransform* property), 409
`FunctionClass` (class in *diofant.core.function*), 126
`FunctionMatrix` (class in *diofant.matrices.expressions*), 503
`futrig()` (in module *diofant.simplify.trigsimp*), 594

G

`G()` (in module *diofant.printing.pretty_symbology*), 562
`g()` (in module *diofant.printing.pretty_symbology*), 562
`GaloisFieldElement` (class in *diofant.domains.finitefield*), 432
`gamma` (class in *diofant.functions.special.gamma_functions*), 319
`gauss_chebyshev_t()` (in module *diofant.integrals.quadrature*), 414
`gauss_chebyshev_u()` (in module *diofant.integrals.quadrature*), 415
`gauss_gen_laguerre()` (in module *diofant.integrals.quadrature*), 413
`gauss_hermite()` (in module *diofant.integrals.quadrature*), 412
`gauss_jacobi()` (in module *diofant.integrals.quadrature*), 416
`gauss_laguerre()` (in module *diofant.integrals.quadrature*), 411
`gaussian_legendre()` (in module *diofant.integrals.quadrature*), 411
`gaussian_reduce()` (in module *diofant.solvers.diophantine*), 632
`gcd()` (*diofant.core.numbers.Number* method), 85
`gcd()` (*diofant.core.numbers.Rational* method), 89
`gcd()` (*diofant.domains.field.Field* method), 427
`gcd()` (*diofant.polys.monomials.Monomial* method), 540
`gcd()` (*diofant.polys.polytools.Poly* method), 515
`gcd()` (in module *diofant.polys.polytools*), 515
`gcd_terms()` (in module *diofant.core.exprtools*), 141
`gcdex()` (*diofant.polys.polytools.Poly* method), 515
`gcdex()` (*diofant.polys.rings.PolyElement* method), 761
`gcdex()` (*diofant.polys.rings.PolynomialRing* method), 430
`gcdex()` (in module *diofant.polys.polytools*), 531
`Ge` (in module *diofant.core.relational*), 108
`gegenbauer` (class in *diofant.functions.special.polynomials*), 377
`gegenbauer_poly()` (in module *diofant.polys.orthopolys*), 543
`gen` (*diofant.polys.polytools.Poly* property), 515
`generate()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 181
`generate_dimino()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 181
`generate_gray()` (*diofant.combinatorics.graycode.GrayCode* method), 209
`generate_schreier_sims()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 182
`generators` (*diofant.combinatorics.perm_groups.PermutationGroup* property), 182
`GeneratorsError`, 785
`GeneratorsNeededError`, 785
`get()` (*diofant.core.containers.Dict* method), 140
`get_adjacency_distance()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 182

`fant.combinatorics.permutations.Permutation` (class in `diofant.combinatorics.permutations`), 155
`get_adjacency_matrix()` (`diofant.combinatorics.permutations.Permutation` method), 155
`get_contraction_structure()` (in module `diofant.tensor.index_methods`), 708
`get_default_datatype()` (in module `diofant.utilities.codegen`), 723
`get_diag_blocks()` (`diofant.matrices.matrices.MatrixBase` method), 453
`get_exact()` (`diofant.domains.domain.Domain` method), 427
`get_indices()` (in module `diofant.tensor.index_methods`), 709
`get_interface()` (`diofant.utilities.codegen.FCodeGen` method), 720
`get_modulus()` (`diofant.polys.polytools.Poly` method), 516
`get_period()` (`diofant.functions.special.hyper.meijerg` method), 371
`get_positional_distance()` (`diofant.combinatorics.permutations.Permutation` method), 156
`get_precedence_distance()` (`diofant.combinatorics.permutations.Permutation` method), 156
`get_precedence_matrix()` (`diofant.combinatorics.permutations.Permutation` method), 156
`get_prototype()` (`diofant.utilities.codegen.CCodeGen` method), 718
`get_subset_from_bitstring()` (`diofant.combinatorics.graycode` method), 212
`get_symmetric_group_sgs()` (in module `diofant.combinatorics.tensor_can`), 228
`getn()` (`diofant.core.expr.Expr` method), 69
`get0()` (`diofant.core.add.Add` method), 105
`get0()` (`diofant.core.expr.Expr` method), 69
`GMPYFiniteField` (class in `diofant.domains.finitefield`), 431
`GMPYIntegerRing` (class in `diofant.domains.integerring`), 432
`GMPYRationalField` (class in `diofant.domains.rationalfield`), 432
`GoldenRatio` (class in `diofant.core.numbers`), 98
`gospers_normal()` (in module `diofant.concrete.gospers`), 272
`gospers_sum()` (in module `diofant.concrete.gospers`), 273
`gospers_term()` (in module `diofant.concrete.gospers`), 273
`GradedLexOrder` (class in `diofant.polys.orderings`), 540
`GramSchmidt()` (in module `diofant.matrices.dense`), 476
`gray_to_bin()` (`diofant.combinatorics.graycode` method), 211
`GrayCode` (class in `diofant.combinatorics.graycode`), 208
`graycode_subsets()` (`diofant.combinatorics.graycode` method), 212
`GreaterThan` (class in `diofant.core.relational`), 110
`greek_letters` (in module `diofant.printing.pretty_symbology`), 562
`groebner()` (in module `diofant.polys.groebnertools`), 769
`groebner()` (in module `diofant.polys.polytools`), 531
`groebner_gcd()` (in module `diofant.polys.groebnertools`), 769
`groebner_lcm()` (in module `diofant.polys.groebnertools`), 769
`GroebnerBasis` (class in `diofant.polys.polytools`), 505
`group()` (in module `diofant.utilities.iterables`), 733
`Gt` (in module `diofant.core.relational`), 107

H

`H` (`diofant.matrices.matrices.MatrixBase` property), 441
`Half` (class in `diofant.core.numbers`), 93
`half_gcdex()` (`diofant.domains.ring.CommutativeRing` method), 428
`half_gcdex()` (`diofant.polys.polytools.Poly` method), 516
`half_gcdex()` (`diofant.polys.rings.PolyElement` method), 761
`half_gcdex()` (`diofant.polys.rings.PolynomialRing` method), 430
`half_gcdex()` (in module `diofant.polys.polytools`), 532
`hankell` (class in `diofant.functions.special.bessel`), 353

[hankel2](#) (class in [diofant.functions.special.bessel](#)), 354
[hankel_transform\(\)](#) (in module [diofant.integrals.transforms](#)), 397
[HankelTransform](#) (class in [diofant.integrals.transforms](#)), 410
[harmonic](#) (class in [diofant.functions.combinatorial.numbers](#)), 313
[has\(\)](#) ([diofant.core.basic.Basic](#) method), 48
[has\(\)](#) ([diofant.matrices.matrices.MatrixBase](#) method), 453
[has\(\)](#) ([diofant.matrices.sparse.SparseMatrixBase](#) method), 493
[has_only_gens\(\)](#) ([diofant.polys.polytools.Poly](#) method), 516
[Heaviside](#) (class in [diofant.functions.special.delta_functions](#)), 319
[height\(\)](#) ([diofant.printing.stringpict.stringPict](#) method), 564
[hermite](#) (class in [diofant.functions.special.polynomials](#)), 382
[hermite_poly\(\)](#) (in module [diofant.polys.orthopolys](#)), 543
[hessian\(\)](#) (in module [diofant.matrices.dense](#)), 475
[heurisch\(\)](#) (in module [diofant.integrals.heurisch](#)), 403
[heurisch_wrapper\(\)](#) (in module [diofant.integrals.heurisch](#)), 404
[HeuristicGCDFailedError](#), 785
[hobj\(\)](#) (in module [diofant.printing.pretty_symbology](#)), 563
[holzer\(\)](#) (in module [diofant.solvers.diophantine](#)), 633
[homogeneous_order\(\)](#) (in module [diofant.solvers.ode](#)), 641
[HomomorphismFailedError](#), 785
[horner\(\)](#) (in module [diofant.polys.polyfuncs](#)), 537
[hstack\(\)](#) ([diofant.matrices.matrices.MatrixBase](#) class method), 453
[hyper](#) (class in [diofant.functions.special.hyper](#)), 367
[HyperbolicFunction](#) (class in [diofant.functions.elementary.hyperbolic](#)), 291
[hyperexpand\(\)](#) (in module [diofant.simplify.hyperexpand](#)), 603
[hypersimilar\(\)](#) (in module [diofant.simplify.simplify](#)), 585
[hypersimp\(\)](#) (in module [diofant.simplify.simplify](#)), 584
[I](#)
[Identity](#) (class in [diofant.matrices.expressions](#)), 503
[IdentityFunction](#) (class in [diofant.functions.elementary.miscellaneous](#)), 298
[Idx](#) (class in [diofant.tensor.indexed](#)), 703
[im](#) (class in [diofant.functions.elementary.complexes](#)), 275
[ImageSet](#) (class in [diofant.sets.fancysets](#)), 579
[imageset\(\)](#) (in module [diofant.sets.sets](#)), 572
[ImaginaryUnit](#) (class in [diofant.core.numbers](#)), 96
[ImmutableDenseNDimArray](#) (class in [diofant.tensor.array](#)), 700
[ImmutableMatrix](#) (class in [diofant.matrices.immutable](#)), 498
[ImmutableSparseMatrix](#) (class in [diofant.matrices.immutable](#)), 496
[ImmutableSparseNDimArray](#) (class in [diofant.tensor.array](#)), 700
[implemented_function\(\)](#) (in module [diofant.utilities.lambdify](#)), 744
[implicit_application\(\)](#) (in module [diofant.parsing.sympy_parser](#)), 750
[implicit_multiplication\(\)](#) (in module [diofant.parsing.sympy_parser](#)), 750
[implicit_multiplication_application\(\)](#) (in module [diofant.parsing.sympy_parser](#)), 751
[Implies](#) (class in [diofant.logic.boolalg](#)), 421
[indent_code\(\)](#) ([diofant.printing.ccode.CCodePrinter](#) method), 552
[indent_code\(\)](#) ([diofant.printing.fcode.FCodePrinter](#) method), 555
[independent_sets](#) ([diofant.polys.polytools.GroebnerBasis](#) property), 505
[index\(\)](#) ([diofant.combinatorics.permutations.Permutation](#) method), 157
[index\(\)](#) ([diofant.concrete.expr_with_intlimits.ExprWithIntLimits](#) method), 270
[index\(\)](#) ([diofant.core.containers.Tuple](#) method), 139
[index\(\)](#) ([diofant.polys.rings.PolynomialRing](#) method), 430

- IndexConformanceExceptionError, 708
- Indexed (class in diofant.tensor.indexed), 705
- IndexedBase (class in diofant.tensor.indexed), 706
- IndexExceptionError, 705
- indices (diofant.tensor.indexed.Indexed property), 705
- indices_contain_equal_information (diofant.functions.special.tensor_functions.KroneckerDelta property), 389
- inf (diofant.sets.sets.Interval property), 573
- inf (diofant.sets.sets.Set property), 568
- infinitesimals() (in module diofant.solvers.ode), 641
- Infinity (class in diofant.core.numbers), 94
- init_printing() (in module diofant.interactive.printing), 566
- inject() (diofant.domains.compositedomain.CompositeDomain method), 429
- inject() (diofant.domains.simplesdomain.SimpleDomain property), 428
- inject() (diofant.polys.polytools.Poly method), 516
- InputArgument (class in diofant.utilities.codegen), 720
- Integer (class in diofant.core.numbers), 89
- integer_digits() (in module diofant.core.numbers), 92
- integer_nthroot() (in module diofant.core.power), 101
- integer_rational_reconstruction() (in module diofant.ntheory.modular), 247
- IntegerDivisionWrapper (class in diofant.interactive.session), 566
- IntegerModRing (class in diofant.domains), 429
- IntegerPartition (class in diofant.combinatorics.partitions), 144
- IntegerRing (class in diofant.domains), 429
- Integers (class in diofant.sets.fancysets), 579
- Integral (class in diofant.integrals.integrals), 405
- Integral.is_commutative (in module diofant.integrals.integrals), 405
- IntegralTransform (class in diofant.integrals.transforms), 408
- integrand() (diofant.functions.special.hyper.meijerg method), 372
- integrate() (diofant.core.expr.Expr method), 69
- integrate() (diofant.matrices.immutable.ImmutableMatrix method), 499
- integrate() (diofant.matrices.matrices.MatrixBase method), 453
- integrate() (diofant.polys.polytools.Poly method), 516
- integrate() (diofant.polys.rings.PolyElement method), 761
- integrate() (in module diofant.integrals.integrals), 399
- Intersection (class in diofant.sets.sets), 576
- intersection() (diofant.sets.sets.Set method), 568
- Interval (class in diofant.sets.sets), 572
- interval (diofant.polys.rootoftools.RootOf property), 541
- inv() (diofant.matrices.matrices.MatrixBase method), 454
- inv_mod() (diofant.matrices.matrices.MatrixBase method), 454
- Inverse (class in diofant.matrices.expressions), 502
- inverse() (diofant.functions.elementary.exponential.log method), 297
- inverse() (diofant.functions.elementary.hyperbolic.acosh method), 293
- inverse() (diofant.functions.elementary.hyperbolic.acoth method), 294
- inverse() (diofant.functions.elementary.hyperbolic.asinh method), 293
- inverse() (diofant.functions.elementary.hyperbolic.atanh method), 293
- inverse() (diofant.functions.elementary.hyperbolic.coth method), 292
- inverse() (diofant.functions.elementary.hyperbolic.sinh method), 291
- inverse() (diofant.functions.elementary.hyperbolic.tanh method), 292
- inverse() (diofant.functions.elementary.trigonometric.acos method), 286
- inverse() (diofant.functions.elementary.trigonometric.acot method), 288
- inverse() (diofant.functions.elementary.trigonometric.acsc method), 289
- inverse() (diofant.functions.elementary.trigonometric.asec method), 287
- inverse() (diofant.functions.elementary.trigonometric.asin method), 285
- inverse() (diofant.functions.elementary.trigonometric.atan method), 288

`inverse()` (`diofant.functions.elementary.trigonometric` module), 283
`inverse()` (`diofant.functions.elementary.trigonometric` module), 282
`inverse_ADJ()` (`diofant.matrices.matrices.MatrixBase` method), 454
`inverse_cosine_transform()` (in module `diofant.integrals.transforms`), 397
`inverse_fourier_transform()` (in module `diofant.integrals.transforms`), 395
`inverse_GE()` (`diofant.matrices.matrices.MatrixBase` method), 454
`inverse_hankel_transform()` (in module `diofant.integrals.transforms`), 398
`inverse_laplace_transform()` (in module `diofant.integrals.transforms`), 394
`inverse_LU()` (`diofant.matrices.matrices.MatrixBase` method), 455
`inverse_mellin_transform()` (in module `diofant.integrals.transforms`), 393
`inverse_sine_transform()` (in module `diofant.integrals.transforms`), 396
`InverseCosineTransform` (class in `diofant.integrals.transforms`), 410
`InverseFourierTransform` (class in `diofant.integrals.transforms`), 410
`InverseHankelTransform` (class in `diofant.integrals.transforms`), 410
`InverseLaplaceTransform` (class in `diofant.integrals.transforms`), 409
`InverseMellinTransform` (class in `diofant.integrals.transforms`), 409
`InverseSineTransform` (class in `diofant.integrals.transforms`), 410
`inversion_vector()` (`diofant.combinatorics.permutations.Permutation` method), 157
`inversions()` (`diofant.combinatorics.permutations.Permutation` method), 158
`invert()` (`diofant.core.expr.Expr` method), 69
`invert()` (`diofant.domains.ring.CommutativeRing` method), 428
`invert()` (`diofant.polys.polytools.Poly` method), 517
`invert()` (in module `diofant.polys.polytools`), 532
`is_abelian` (`diofant.combinatorics.perm_groups.PermutationGroup` property), 182
`is_above_fermi` (`diofant.functions.special.tensor_functions.KroneckerDelta` property), 390
`is_algebraic` (`diofant.core.expr.Expr` property), 69
`is_algebraic_expr()` (`diofant.core.expr.Expr` method), 69
`is_alt_sym()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 182
`is_anti_symmetric()` (`diofant.matrices.matrices.MatrixBase` method), 455
`is_below_fermi` (`diofant.functions.special.tensor_functions.KroneckerDelta` property), 390
`is_closed` (`diofant.sets.sets.Set` property), 568
`is_cnf()` (in module `diofant.logic.boolalg`), 423
`is_commutative` (`diofant.core.expr.Expr` property), 70
`is_comparable` (`diofant.core.expr.Expr` property), 70
`is_complex` (`diofant.core.expr.Expr` property), 70
`is_composite` (`diofant.core.expr.Expr` property), 70
`is_composite` (`diofant.core.numbers.Integer` property), 89
`is_constant()` (`diofant.core.expr.Expr` method), 71
`is_cyclotomic` (`diofant.polys.polytools.Poly` property), 517
`is_diagonal()` (`diofant.matrices.matrices.MatrixBase` method), 456
`is_diagonalizable()` (`diofant.matrices.matrices.MatrixBase` method), 456
`is_disjoint()` (`diofant.polys.rootisolation.ComplexInterval` method), 785
`is_disjoint()` (`diofant.polys.rootisolation.RealInterval` method), 785
`is_disjoint()` (`diofant.sets.sets.Set` method), 569
`is_dnf()` (in module `diofant.logic.boolalg`), 424
`is_Empty` (`diofant.combinatorics.permutations.Permutation` property), 158
`is_evaluated` (`diofant.core.basic.Basic` property), 49
`is_even` (`diofant.combinatorics.permutations.Permutation`

property), 159

`is_even (diofant.core.expr.Expr property)`, 71

`is_even (diofant.core.numbers.Integer property)`, 90

`is_extended_real (diofant.core.expr.Expr property)`, 72

`is_finite (diofant.core.expr.Expr property)`, 72

`is_groebner()` (in module `diofant.polys.groebnertools`), 769

`is_ground (diofant.polys.polytools.Poly property)`, 517

`is_hermitian (diofant.matrices.matrices.MatrixBase property)`, 457

`is_hermitian (diofant.matrices.sparse.SparseMatrixBase property)`, 493

`is_homogeneous (diofant.polys.polytools.Poly property)`, 517

`is_hypergeometric()` (diofant.core.expr.Expr method), 72

`is_Identity (diofant.combinatorics.permutations.Permutation property)`, 158

`is_imaginary (diofant.core.expr.Expr property)`, 72

`is_imaginary (diofant.core.numbers.Integer property)`, 90

`is_infinite (diofant.core.expr.Expr property)`, 73

`is_integer (diofant.core.expr.Expr property)`, 73

`is_irrational (diofant.core.expr.Expr property)`, 73

`is_irreducible (diofant.polys.polytools.Poly property)`, 518

`is_iterable()` (in module `diofant.utilities.iterables`), 734

`is_left_unbounded (diofant.sets.sets.Interval property)`, 573

`is_linear (diofant.polys.polytools.Poly property)`, 518

`is_lower (diofant.matrices.matrices.MatrixBase property)`, 457

`is_lower_hessenberg (diofant.matrices.matrices.MatrixBase property)`, 458

`is_minimal()` (in module `diofant.polys.groebnertools`), 769

`is_multivariate (diofant.polys.polytools.Poly property)`, 518

`is_negative (diofant.core.expr.Expr property)`, 73

`is_nilpotent (diofant.combinatorics.perm_groups.PermutationGroup property)`, 183

`is_nilpotent()` (diofant.matrices.matrices.MatrixBase method), 458

`is_nnf()` (in module `diofant.logic.boolalg`), 424

`is_noninteger (diofant.core.expr.Expr property)`, 73

`is_nonnegative (diofant.core.expr.Expr property)`, 73

`is_nonpositive (diofant.core.expr.Expr property)`, 73

`is_nonzero (diofant.core.expr.Expr property)`, 73

`is_nonzero (diofant.core.numbers.Integer property)`, 90

`is_normal()` (diofant.combinatorics.perm_groups.Permutation method), 183

`is_normal()` (diofant.domains.ring.CommutativeRing method), 428

`is_nthpow_residue()` (in module `diofant.ntheory.residue_ntheory`), 252

`is_number (diofant.concrete.expr_with_limits.ExprWithLimits property)`, 268

`is_number (diofant.core.expr.Expr property)`, 74

`is_odd (diofant.combinatorics.permutations.Permutation property)`, 159

`is_odd (diofant.core.expr.Expr property)`, 74

`is_odd (diofant.core.numbers.Integer property)`, 90

`is_one (diofant.polys.polytools.Poly property)`, 518

`is_only_above_fermi (diofant.functions.special.tensor_functions.KroneckerDelta property)`, 390

`is_only_below_fermi (diofant.functions.special.tensor_functions.KroneckerDelta property)`, 391

`is_open (diofant.sets.sets.Set property)`, 569

`is_polar (diofant.core.expr.Expr property)`, 74

`is_polynomial()` (diofant.core.expr.Expr method), 74

`is_positive (diofant.core.expr.Expr property)`, 75

`is_prime (diofant.core.expr.Expr property)`, 75

`is_prime (diofant.core.numbers.Integer property)`, 90

`is_primitive (diofant.core.expr.Expr property)`, 73

`fant.domains.finitefield.ModularInteger`
`property)`, 432
`is_primitive()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 184
`is_primitive_root()` (in module `diofant.ntheory.residue_ntheory`), 252
`is_proper_subset()` (`diofant.sets.sets.Set` method), 569
`is_proper_superset()` (`diofant.sets.sets.Set` method), 569
`is_quad_residue()` (in module `diofant.ntheory.residue_ntheory`), 253
`is_quadratic` (`diofant.polys.polytools.Poly` property), 518
`is_rational` (`diofant.core.expr.Expr` property), 75
`is_rational_function()` (`diofant.core.expr.Expr` method), 75
`is_real` (`diofant.core.expr.Expr` property), 76
`is_rewritable_or_comparable()` (in module `diofant.polys.groebnertools`), 769
`is_right_unbounded` (`diofant.sets.sets.Interval` property), 573
`is_sequence()` (in module `diofant.utilities.iterables`), 734
`is_simple()` (`diofant.functions.special.delta_functions.DeltaError` method), 318
`is_Singleton` (`diofant.combinatorics.permutations.Permutation` property), 159
`is_solvable` (`diofant.combinatorics.perm_groups.PermutationGroup` property), 184
`is_square` (`diofant.matrices.matrices.MatrixBase` property), 458
`is_square()` (in module `diofant.ntheory.primetest`), 250
`is_squarefree` (`diofant.polys.polytools.Poly` property), 519
`is_subgroup()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 184
`is_subset()` (`diofant.sets.sets.Set` method), 570
`is_superset()` (`diofant.sets.sets.Set` method), 570
`is_symbolic()` (`diofant.matrices.matrices.MatrixBase` method), 459
`is_symmetric()` (`diofant.matrices.matrices.MatrixBase` method), 459
`is_symmetric()` (`diofant.matrices.sparse.SparseMatrixBase` method), 493
`is_term` (`diofant.polys.polytools.Poly` property), 519
`is_transcendental` (`diofant.core.expr.Expr` property), 76
`is_transitive()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 185
`is_trivial` (`diofant.combinatorics.perm_groups.PermutationGroup` property), 185
`is_univariate` (`diofant.polys.polytools.Poly` property), 519
`is_upper` (`diofant.matrices.matrices.MatrixBase` property), 460
`is_upper_hessenberg` (`diofant.matrices.matrices.MatrixBase` property), 460
`is_zero` (`diofant.core.expr.Expr` property), 76
`is_zero` (`diofant.core.numbers.Integer` property), 91
`is_zero` (`diofant.matrices.matrices.MatrixBase` property), 460
`is_zero` (`diofant.polys.polytools.Poly` property), 519
`isdisjoint()` (`diofant.sets.sets.Set` method), 570
`IsomorphismError`, 786
`isprime()` (in module `diofant.ntheory.primetest`), 251
`is_subset()` (`diofant.sets.sets.Set` method), 570
`is_subset()` (`diofant.sets.sets.Set` method), 570
`ITE` (class in `diofant.logic.boolalg`), 422
`items()` (`diofant.core.containers.Dict` method), 140
`iterate_binary()` (`diofant.combinatorics.subsets.Subset` method), 203
`iterate_graycode()` (`diofant.combinatorics.subsets.Subset` method), 203
J
`jacobi` (class in `diofant.functions.special.polynomials`), 374
`jacobi_normalized()` (in module `diofant.functions.special.polynomials`), 376
`jacobi_poly()` (in module `diofant.polys.orthopolys`), 543

jacobi_symbol() (in module diofant.ntheory.residue_ntheory), 253
 jacobian() (diofant.matrices.matrices.MatrixBase method), 461
 jn (class in diofant.functions.special.bessel), 354
 jn_zeros() (in module diofant.functions.special.bessel), 355
 jordan_cell() (in module diofant.matrices.dense), 475
 jordan_cells() (diofant.matrices.matrices.MatrixBase method), 461
 jordan_form() (diofant.matrices.matrices.MatrixBase method), 462
 josephus() (diofant.combinatorics.permutations.Permutation class method), 160
K
 key2bounds() (diofant.matrices.matrices.MatrixBase method), 462
 key2ij() (diofant.matrices.matrices.MatrixBase method), 462
 keys() (diofant.core.containers.Dict method), 140
 killable_index (diofant.functions.special.tensor_functions.KroneckerDelta property), 391
 KroneckerDelta (class in diofant.functions.special.tensor_functions), 388
 ksubsets() (diofant.combinatorics.subsets method), 208
L
 label (diofant.tensor.indexed.Idx property), 704
 label (diofant.tensor.indexed.IndexedBase property), 707
 laguerre (class in diofant.functions.special.polynomials), 383
 laguerre_poly() (in module diofant.polys.orthopolys), 543
 Lambda (class in diofant.core.function), 121
 LambdaPrinter (class in diofant.printing.lambdarepr), 558
 lambdarepr() (in module diofant.printing.lambdarepr), 558
 lambdastr() (in module diofant.utilities.lambdify), 745
 lambdify() (in module diofant.utilities.lambdify), 745
 LambertW (class in diofant.functions.elementary.exponential), 296
 laplace_transform() (in module diofant.integrals.transforms), 394
 LaplaceTransform (class in diofant.integrals.transforms), 409
 latex() (in module diofant.printing.latex), 558
 LatexPrinter (class in diofant.printing.latex), 558
 lbp_cmp() (in module diofant.polys.groebnertools), 769
 lbp_key() (in module diofant.polys.groebnertools), 769
 lbp_mul_term() (in module diofant.polys.groebnertools), 770
 lbp_sub() (in module diofant.polys.groebnertools), 770
 LC() (diofant.polys.polytools.Poly method), 507
 LC() (in module diofant.polys.polytools), 506
 lcm() (diofant.core.numbers.Number method), 85
 lcm() (diofant.core.numbers.Rational method), 89
 lcm() (diofant.domains.ring.CommutativeRing method), 428
 lcm() (diofant.polys.monomials.Monomial method), 540
 lcm() (diofant.polys.polytools.Poly method), 519
 lcm() (in module diofant.polys.polytools), 533
 ldescent() (in module diofant.solvers.diophantine), 632
 LDLdecomposition() (diofant.matrices.matrices.MatrixBase method), 441
 LDLdecomposition() (diofant.matrices.sparse.SparseMatrixBase method), 490
 LDLsolve() (diofant.matrices.matrices.MatrixBase method), 442
 Le (in module diofant.core.relational), 107
 leading_expv() (diofant.polys.rings.PolyElement method), 761
 leading_term() (diofant.polys.rings.PolyElement method), 761
 leadterm() (in module dio-

- `fant.calculus.gruntz`), 788
- `left` (`diofant.sets.sets.Interval` property), 573
- `left()` (`diofant.printing.stringpict.stringPict` method), 564
- `left_open` (`diofant.sets.sets.Interval` property), 574
- `legendre` (class in `diofant.functions.special.polynomials`), 381
- `legendre_poly()` (in module `diofant.polys.orthopolys`), 543
- `legendre_symbol()` (in module `diofant.ntheory.residue_ntheory`), 254
- `length()` (`diofant.combinatorics.permutations.Permutation` method), 160
- `length()` (`diofant.polys.polytools.Poly` method), 520
- `lerchphi` (class in `diofant.functions.special.zeta_functions`), 366
- `LessThan` (class in `diofant.core.relational`), 112
- `LeviCivita` (class in `diofant.functions.special.tensor_functions`), 388
- `LexOrder` (class in `diofant.polys.orderings`), 540
- `lhs` (`diofant.core.relational.Relational` property), 108
- `Li` (class in `diofant.functions.special.error_functions`), 344
- `li` (class in `diofant.functions.special.error_functions`), 343
- `lie_heuristic_abaco1_product()` (in module `diofant.solvers.ode`), 665
- `lie_heuristic_abaco1_simple()` (in module `diofant.solvers.ode`), 664
- `lie_heuristic_abaco2_similar()` (in module `diofant.solvers.ode`), 666
- `lie_heuristic_abaco2_unique_unknown()` (in module `diofant.solvers.ode`), 667
- `lie_heuristic_bivariate()` (in module `diofant.solvers.ode`), 665
- `lie_heuristic_chi()` (in module `diofant.solvers.ode`), 665
- `lie_heuristic_function_sum()` (in module `diofant.solvers.ode`), 667
- `lie_heuristic_linear()` (in module `diofant.solvers.ode`), 668
- `Limit` (class in `diofant.calculus.limits`), 752
- `limit()` (`diofant.core.expr.Expr` method), 76
- `limit()` (`diofant.matrices.immutable.ImmutableMatrix` method), 499
- `limit()` (`diofant.matrices.matrices.MatrixBase` method), 463
- `limit()` (in module `diofant.calculus.limits`), 753
- `limit_denominator()` (`diofant.core.numbers.Rational` method), 89
- `limitinf()` (in module `diofant.calculus.gruntz`), 788
- `limits` (`diofant.concrete.expr_with_limits.ExprWithLimits` property), 268
- `list()` (`diofant.combinatorics.permutations.Cycle` method), 168
- `list()` (`diofant.combinatorics.permutations.Permutation` method), 160
- `list2numpy()` (in module `diofant.matrices.dense`), 477
- `list_visitor()` (in module `diofant.utilities.enumerative`), 726
- `liupc()` (`diofant.matrices.sparse.SparseMatrixBase` method), 493
- `LM()` (`diofant.polys.polytools.Poly` method), 507
- `LM()` (in module `diofant.polys.polytools`), 506
- `log` (class in `diofant.functions.elementary.exponential`), 296
- `logcombine()` (in module `diofant.simplify.simplify`), 586
- `loggamma` (class in `diofant.functions.special.gamma_functions`), 321
- `Lopen()` (`diofant.sets.sets.Interval` class method), 573
- `lower` (`diofant.tensor.indexed.Idx` property), 704
- `lower_central_series()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 186
- `lower_triangular_solve()` (`diofant.matrices.matrices.MatrixBase` method), 463
- `lowergamma` (class in `diofant.functions.special.gamma_functions`), 326
- `Lt` (in module `diofant.core.relational`), 107
- `LT()` (`diofant.polys.polytools.Poly` method), 508
- `LT()` (in module `diofant.polys.polytools`), 506
- `lucas` (class in `diofant.functions.combinatorial.numbers`), 315

LUdecomposition() (diofant.matrices.matrices.MatrixBase method), 442
LUdecomposition_Simple() (diofant.matrices.matrices.MatrixBase method), 443
LUdecompositionFF() (diofant.matrices.matrices.MatrixBase method), 442
LUsolve() (diofant.matrices.matrices.MatrixBase method), 443
M
make_perm() (diofant.combinatorics.perm_groups.permutation_group method), 186
make_property() (in module diofant.core.assumptions), 45
make_routine() (in module diofant.utilities.codegen), 723
ManagedProperties (class in diofant.core.assumptions), 44
MatAdd (class in diofant.matrices.expressions), 501
match() (diofant.core.basic.Basic method), 49
mathematica() (in module diofant.parsing.mathematica), 749
mathematica_code() (in module diofant.printing.mathematica), 557
mathml() (in module diofant.printing.mathml), 560
mathml_tag() (diofant.printing.mathml.MathMLPrinter method), 560
MathMLPrinter (class in diofant.printing.mathml), 560
MatMul (class in diofant.matrices.expressions), 502
MatPow (class in diofant.matrices.expressions), 502
Matrix (class in diofant.matrices), 432
matrix2numpy() (in module diofant.matrices.dense), 477
matrix_fgln() (in module diofant.polys.groebnertools), 770
matrix_multiply_elementwise() (in module diofant.matrices.dense), 473
MatrixBase (class in diofant.matrices.matrices), 440
MatrixError (class in diofant.matrices.matrices), 473
MatrixExpr (class in diofant.matrices.expressions), 500
MatrixSymbol (class in diofant.matrices.expressions), 501
Max (class in diofant.functions.elementary.miscellaneous), 299
max() (diofant.combinatorics.permutations.Permutation method), 161
max_div (diofant.combinatorics.perm_groups.PermutationGroup property), 187
maximize() (in module diofant.calculus.optimization), 753
MCodePrinter (class in diofant.printing.mathematica), 557
measure (diofant.sets.Set property), 570
meijerg (class in diofant.functions.special.hyper), 369
mellin_transform() (in module diofant.integrals.transforms), 393
MellinTransform (class in diofant.integrals.transforms), 409
merge_solution() (in module diofant.solvers.diophantine), 629
Min (class in diofant.functions.elementary.miscellaneous), 299
min() (diofant.combinatorics.permutations.Permutation method), 161
minimal_block() (diofant.combinatorics.perm_groups.PermutationGroup method), 187
minimal_polynomial() (in module diofant.polys.numberfields), 539
minimize() (in module diofant.calculus.optimization), 753
minlex() (in module diofant.utilities.iterables), 735
minorEntry() (diofant.matrices.matrices.MatrixBase method), 463
minorMatrix() (diofant.matrices.matrices.MatrixBase method), 463
minpoly_groebner() (in module diofant.polys.numberfields), 771
minsolve_linear_system() (in module diofant.solvers.solvers), 606
mobius (class in diofant.ntheory.residue_ntheory), 254
Mod (class in diofant.core.mod), 106
mod_inverse() (in module diofant.core.numbers), 91
modgcd() (in module diofant.polys.modulargcd), 782
ModularGCDFailedError, 786

ModularInteger (class in diofant.domains.finitefield), 432
 module
 diofant, 41
 diofant.calculus, 752
 diofant.calculus.gruntz, 787
 diofant.calculus.limits, 752
 diofant.calculus.optimization, 753
 diofant.calculus.order, 753
 diofant.calculus.residues, 755
 diofant.calculus.singularities, 752
 diofant.combinatorics.generators, 169
 diofant.combinatorics.graycode, 208
 diofant.combinatorics.group_constructs, 221
 diofant.combinatorics.named_groups, 212
 diofant.combinatorics.partitions, 142
 diofant.combinatorics.perm_groups, 170
 diofant.combinatorics.permutations, 147
 diofant.combinatorics.polyhedron, 196
 diofant.combinatorics.prufer, 199
 diofant.combinatorics.subsets, 202
 diofant.combinatorics.tensor_can, 223
 diofant.combinatorics.testutil, 221
 diofant.combinatorics.util, 215
 diofant.config, 41
 diofant.core, 41
 diofant.core.add, 104
 diofant.core.assumptions, 44
 diofant.core.basic, 46
 diofant.core.cache, 45
 diofant.core.compatibility, 140
 diofant.core.containers, 139
 diofant.core.core, 55
 diofant.core.evalf, 138
 diofant.core.evaluate, 56
 diofant.core.expr, 57
 diofant.core.exprtools, 141
 diofant.core.function, 121
 diofant.core.mod, 106
 diofant.core.mul, 101
 diofant.core.multidimensional, 120
 diofant.core.numbers, 85
 diofant.core.power, 99
 diofant.core.relational, 107
 diofant.core.singleton, 55
 diofant.core.symbol, 80
 diofant.core.sympify, 41
 diofant.domains, 426
 diofant.functions, 274
 diofant.functions.special.bessel, 351
 diofant.functions.special.beta_functions, 328
 diofant.functions.special.elliptic_integrals, 372
 diofant.functions.special.error_functions, 329
 diofant.functions.special.gamma_functions, 319
 diofant.functions.special.polynomials, 374
 diofant.functions.special.zeta_functions, 363
 diofant.integrals, 392
 diofant.integrals.meijerint_doc, 809
 diofant.integrals.quadrature, 411
 diofant.integrals.transforms, 393
 diofant.interactive, 566
 diofant.interactive.printing, 566
 diofant.interactive.session, 566
 diofant.logic, 417
 diofant.matrices, 432
 diofant.matrices.dense, 479
 diofant.matrices.expressions, 500
 diofant.matrices.expressions.blockmatrix, 503
 diofant.matrices.immutable, 498
 diofant.matrices.matrices, 432
 diofant.matrices.sparse, 487
 diofant.ntheory.continued_fraction, 257
 diofant.ntheory.egyptian_fraction, 259
 diofant.ntheory.factor_, 235
 diofant.ntheory.generate, 228
 diofant.ntheory.modular, 246
 diofant.ntheory.multinomial, 249
 diofant.ntheory.partitions_, 250
 diofant.ntheory.primetest, 250
 diofant.ntheory.residue_ntheory, 252
 diofant.parsing, 748
 diofant.polys, 505
 diofant.polys.constructor, 539
 diofant.polys.euclidtools, 764
 diofant.polys.factorization_alg_field, 771
 diofant.polys.factortools, 767
 diofant.polys.groebnertools, 767

[diofant.polys.modulargcd](#), 775
[diofant.polys.monomials](#), 540
[diofant.polys.numberfields](#), 539
[diofant.polys.orderings](#), 540
[diofant.polys.orthopolys](#), 543
[diofant.polys.partfrac](#), 545
[diofant.polys.polyerrors](#), 785
[diofant.polys.polyfuncs](#), 537
[diofant.polys.polyoptions](#), 785
[diofant.polys.polyroots](#), 542
[diofant.polys.polytools](#), 505
[diofant.polys.rationaltools](#), 544
[diofant.polys.rootisolation](#), 784
[diofant.polys.rootoftools](#), 540
[diofant.polys.specialpolys](#), 543
[diofant.polys.sqfreetools](#), 785
[diofant.printing](#), 548
[diofant.printing.ccode](#), 551
[diofant.printing.codeprinter](#), 562
[diofant.printing.conventions](#), 562
[diofant.printing.fcode](#), 554
[diofant.printing.lambdarepr](#), 558
[diofant.printing.latex](#), 558
[diofant.printing.mathematica](#), 557
[diofant.printing.mathml](#), 560
[diofant.printing.precedence](#), 562
[diofant.printing.pretty](#), 551
[diofant.printing.pretty_symbolology](#), 562
[diofant.printing.printer](#), 548
[diofant.printing.python](#), 561
[diofant.printing.repr](#), 561
[diofant.printing.str](#), 561
[diofant.printing.stringpict](#), 563
[diofant.sets.fancysets](#), 578
[diofant.sets.sets](#), 567
[diofant.simplify.combsimp](#), 599
[diofant.simplify.cse_main](#), 601
[diofant.simplify.epathtools](#), 604
[diofant.simplify.fu](#), 595
[diofant.simplify.hyperexpand](#), 603
[diofant.simplify.hyperexpand_doc](#), 797
[diofant.simplify.powsimp](#), 596
[diofant.simplify.radsimp](#), 587
[diofant.simplify.ratsimp](#), 593
[diofant.simplify.sqrtdenest](#), 599
[diofant.simplify.traversaltools](#), 604
[diofant.simplify.trigsimp](#), 593
[diofant.solvers](#), 606
[diofant.solvers.deutils](#), 696
[diofant.solvers.diophantine](#), 610
[diofant.solvers.inequalities](#), 610

[diofant.solvers.ode](#), 680
[diofant.solvers.pde](#), 695
[diofant.solvers.polysys](#), 608
[diofant.solvers.recurr](#), 684
[diofant.solvers.solvers](#), 606
[diofant.solvers.utils](#), 695
[diofant.tensor](#), 696
[diofant.tensor.array](#), 697
[diofant.tensor.index_methods](#), 708
[diofant.tensor.indexed](#), 702
[diofant.utilities](#), 710
[diofant.utilities.autowrap](#), 711
[diofant.utilities.codegen](#), 716
[diofant.utilities.decorator](#), 725
[diofant.utilities.enumerative](#), 725
[diofant.utilities.iterables](#), 731
[diofant.utilities.lambdify](#), 744
[diofant.utilities.memoization](#), 747
[diofant.utilities.misc](#), 747
[diofant.utilities.randtest](#), 747
[monic\(\)](#) (*diofant.polys.polytools.Poly* method), 520
[monic\(\)](#) (*diofant.polys.rings.PolyElement* method), 762
[monic\(\)](#) (in module *diofant.polys.polytools*), 533
[Monomial](#) (class in *diofant.polys.monomials*), 540
[monoms\(\)](#) (*diofant.polys.polytools.Poly* method), 520
[mr\(\)](#) (in module *diofant.nttheory.primetest*), 251
[mrv\(\)](#) (in module *diofant.calculus.gruntz*), 788
[mrv_max\(\)](#) (in module *diofant.calculus.gruntz*), 788
[Mul](#) (class in *diofant.core.mul*), 101
[mul_inv\(\)](#) (*diofant.combinatorics.permutations.Permutation* method), 161
[multinomial_coefficients\(\)](#) (in module *diofant.nttheory.multinomial*), 249
[multinomial_coefficients_iterator\(\)](#) (in module *diofant.nttheory.multinomial*), 250
[multiplicity\(\)](#) (in module *diofant.nttheory.factor_*), 240
[multiply\(\)](#) (*diofant.matrices.matrices.MatrixBase* method), 463
[multiply\(\)](#) (*diofant.matrices.sparse.SparseMatrixBase* method), 494
[multiply_elementwise\(\)](#) (*diofant.matrices.matrices.MatrixBase* method), 463

[multiset\(\)](#) (in module [diofant.utilities.iterables](#)), 735
[multiset_combinations\(\)](#) (in module [diofant.utilities.iterables](#)), 735
[multiset_partitions\(\)](#) (in module [diofant.utilities.iterables](#)), 736
[multiset_partitions_taocp\(\)](#) (in module [diofant.utilities.enumerative](#)), 725
[multiset_permutations\(\)](#) (in module [diofant.utilities.iterables](#)), 737
[MultisetPartitionTraverser](#) (class in [diofant.utilities.enumerative](#)), 726
[MultivariatePolynomialError](#), 786
[MutableDenseMatrix](#) (class in [diofant.matrices.dense](#)), 482
[MutableDenseNDimArray](#) (class in [diofant.tensor.array](#)), 700
[MutableMatrix](#) (in module [diofant.matrices.dense](#)), 479
[MutableSparseMatrix](#) (class in [diofant.matrices.sparse](#)), 487
[MutableSparseNDimArray](#) (class in [diofant.tensor.array](#)), 700
N
[n](#) ([diofant.combinatorics.graycode.GrayCode](#) property), 209
[N\(\)](#) (in module [diofant.core.evalf](#)), 139
[n_order\(\)](#) (in module [diofant.ntheory.residue_ntheory](#)), 255
[NaN](#) (class in [diofant.core.numbers](#)), 94
[Nand](#) (class in [diofant.logic.boolalg](#)), 421
[nargs](#) ([diofant.core.function.FunctionClass](#) property), 126
[Naturals](#) (class in [diofant.sets.fancysets](#)), 578
[Naturals0](#) (class in [diofant.sets.fancysets](#)), 578
[Ne](#) (in module [diofant.core.relational](#)), 107
[NegativeInfinity](#) (class in [diofant.core.numbers](#)), 95
[NegativeOne](#) (class in [diofant.core.numbers](#)), 93
[new\(\)](#) ([diofant.polys.polytools.Poly](#) class method), 520
[new\(\)](#) ([diofant.polys.rootoftools.RootSum](#) class method), 541
[next\(\)](#) ([diofant.combinatorics.graycode.GrayCode](#) method), 209
[next\(\)](#) ([diofant.combinatorics.prufer.Prufer](#) method), 199
[next\(\)](#) ([diofant.printing.stringpict.stringPict](#) static method), 564
[next_binary\(\)](#) ([diofant.combinatorics.subsets.Subset](#) method), 203
[next_gray\(\)](#) ([diofant.combinatorics.subsets.Subset](#) method), 204
[next_lex\(\)](#) ([diofant.combinatorics.partitions.IntegerPartitions](#) method), 145
[next_lex\(\)](#) ([diofant.combinatorics.permutations.Permutation](#) method), 161
[next_lexicographic\(\)](#) ([diofant.combinatorics.subsets.Subset](#) method), 204
[next_nonlex\(\)](#) ([diofant.combinatorics.permutations.Permutation](#) method), 162
[next_trotterjohnson\(\)](#) ([diofant.combinatorics.permutations.Permutation](#) method), 162
[nextprime\(\)](#) (in module [diofant.ntheory.generate](#)), 231
[nfloat\(\)](#) (in module [diofant.core.function](#)), 138
[nnz\(\)](#) ([diofant.matrices.sparse.SparseMatrixBase](#) method), 494
[no_attrs_in_subclass](#) (class in [diofant.utilities.decorator](#)), 725
[nodes](#) ([diofant.combinatorics.prufer.Prufer](#) property), 199
[NonSquareMatrixError](#) (class in [diofant.matrices.matrices](#)), 473
[Nor](#) (class in [diofant.logic.boolalg](#)), 421
[norm\(\)](#) ([diofant.matrices.matrices.MatrixBase](#) method), 464
[normal\(\)](#) ([diofant.core.expr.Expr](#) method), 76
[normal_closure\(\)](#) ([diofant.combinatorics.perm_groups.PermutationGroup](#) method), 188
[normalized\(\)](#) ([diofant.matrices.matrices.MatrixBase](#) method), 464
[Not](#) (class in [diofant.logic.boolalg](#)), 420
[NotAlgebraicError](#), 786
[NotInvertibleError](#), 786
[NotIterable](#) (class in [diofant.utilities.iterables](#)), 731
[NotReversibleError](#), 786
[partitions\(\)](#) (in module [diofant.ntheory.partitions](#)), 250
[nroots\(\)](#) ([diofant.polys.polytools.Poly](#) method), 520
[nroots\(\)](#) (in module [diofant.polys.polytools](#)), 533
[nseries\(\)](#) ([diofant.core.expr.Expr](#) method),

77
nsimplify() (diofant.core.expr.Expr method), 78
nsimplify() (in module diofant.simplify.simplify), 585
nthroot() (in module diofant.simplify.simplify), 583
nthroot_mod() (in module diofant.ntheory.residue_ntheory), 255
nu (diofant.functions.special.hyper.meijerg property), 372
nullspace() (diofant.matrices.matrices.MatrixBase method), 465
Number (class in diofant.core.numbers), 85
numbered_symbols() (in module diofant.utilities.iterables), 737
NumberSymbol (class in diofant.core.numbers), 91
O
0 (in module diofant.calculus.order), 753
OctaveCodeGen (class in diofant.utilities.codegen), 720
ode_1st_exact() (in module diofant.solvers.ode), 647
ode_1st_homogeneous_coeff_best() (in module diofant.solvers.ode), 648
ode_1st_homogeneous_coeff_subs_dep_div_indep() (in module diofant.solvers.ode), 649
ode_1st_homogeneous_coeff_subs_indep_div_dep() (in module diofant.solvers.ode), 650
ode_1st_linear() (in module diofant.solvers.ode), 651
ode_1st_power_series() (in module diofant.solvers.ode), 662
ode_2nd_power_series_ordinary() (in module diofant.solvers.ode), 662
ode_2nd_power_series_regular() (in module diofant.solvers.ode), 663
ode_almost_linear() (in module diofant.solvers.ode), 658
ode_Bernoulli() (in module diofant.solvers.ode), 652
ode_lie_group() (in module diofant.solvers.ode), 661
ode_linear_coefficients() (in module diofant.solvers.ode), 659
ode_Liouville() (in module diofant.solvers.ode), 653
ode_nth_linear_constant_coeff_homogeneous() (in module diofant.solvers.ode), 654
ode_nth_linear_constant_coeff_undetermined_coefficients() (in module diofant.solvers.ode), 655
ode_nth_linear_constant_coeff_variation_of_parameters() (in module diofant.solvers.ode), 656
ode_order() (in module diofant.solvers.deutils), 696
ode_Riccati_special_minus2() (in module diofant.solvers.ode), 654
ode_separable() (in module diofant.solvers.ode), 657
ode_separable_reduced() (in module diofant.solvers.ode), 660
ode_sol_simplicity() (in module diofant.solvers.ode), 646
odesimp() (in module diofant.solvers.ode), 643
One (class in diofant.core.numbers), 92
ones() (in module diofant.matrices.dense), 473
open() (diofant.sets.sets.Interval class method), 574
OperationNotSupportedError, 786
opt_cse() (in module diofant.simplify.cse_main), 602
OptionError, 786
Or (class in diofant.logic.boolalg), 419
orbit() (diofant.combinatorics.perm_groups.PermutationGroup method), 188
orbit_rep() (diofant.combinatorics.perm_groups.PermutationGroup method), 189
orbit_transversal() (diofant.combinatorics.perm_groups.PermutationGroup method), 189
orbits() (diofant.combinatorics.perm_groups.PermutationGroup method), 189
Order (class in diofant.calculus.order), 754
Order (class in diofant.polys.polyoptions), 785
order (diofant.functions.special.bessel.BesselBase property), 351
order() (diofant.combinatorics.perm_groups.PermutationGroup method), 190
order() (diofant.combinatorics.permutations.Permutation method), 162
ordered() (in module diofant.utilities.iterables), 737
ordered_partitions() (in module diofant.utilities.iterables), 738
OutputArgument (class in diofant.utilities.codegen), 721
P
parallel_poly_from_expr() (in module diofant.polys.polytools), 533
parametrize_ternary_quadratic() (in module diofant.solvers.diophantine), 631

`parens()` (*diofant.printing.stringpict.stringPict* method), 564
`parity()` (*diofant.combinatorics.permutations* method), 163
`parse_expr()` (in module *diofant.parsing.sympy_parser*), 748
`parse_maxima()` (in module *diofant.parsing.maxima*), 749
`Partition` (class in *diofant.combinatorics.partitions*), 142
`partition` (*diofant.combinatorics.partitions.Partition* property), 143
`partition()` (in module *diofant.solvers.diophantine*), 626
`partitions()` (in module *diofant.utilities.iterables*), 740
`pde_1st_linear_constant_coeff()` (in module *diofant.solvers.pde*), 693
`pde_1st_linear_constant_coeff_homogeneous()` (in module *diofant.solvers.pde*), 692
`pde_1st_linear_variable_coeff()` (in module *diofant.solvers.pde*), 694
`pde_separate()` (in module *diofant.solvers.pde*), 688
`pde_separate_add()` (in module *diofant.solvers.pde*), 688
`pde_separate_mul()` (in module *diofant.solvers.pde*), 689
`pdsolve()` (in module *diofant.solvers.pde*), 689
`per()` (*diofant.polys.polytools.Poly* method), 521
`perfect_power()` (in module *diofant.ntheory.factor_*), 240
`periodic_argument` (class in *diofant.functions.elementary.complexes*), 279
`Permutation` (class in *diofant.combinatorics.permutations*), 147
`PermutationGroup` (class in *diofant.combinatorics.perm_groups*), 170
`permute_signs()` (in module *diofant.utilities.iterables*), 741
`permuteBkwd()` (*diofant.matrices.matrices.MatrixBase* method), 465
`permutedims()` (in module *diofant.tensor.array*), 700
`permuteFwd()` (*diofant.matrices.matrices.MatrixBase* method), 465
`pgroup` (*diofant.combinatorics.polyhedron.Polyhedron* property), 197
`Pi` (class in *diofant.core.numbers*), 97
`Piecewise` (class in *diofant.functions.elementary.piecewise*), 297
`piecewise_fold()` (in module *diofant.functions.elementary.piecewise*), 298
`pinv()` (*diofant.matrices.matrices.MatrixBase* method), 465
`pinv_solve()` (*diofant.matrices.matrices.MatrixBase* method), 466
`point` (*diofant.core.function.Subs* property), 129
`pointwise_stabilizer()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 190
`pol_lift` (class in *diofant.functions.elementary.complexes*), 278
`PoleError` (class in *diofant.core.function*), 133
`PolificationFailedError`, 786
`pollard_pml()` (in module *diofant.ntheory.factor_*), 241
`pollard_rho()` (in module *diofant.ntheory.factor_*), 242
`Poly` (class in *diofant.polys.polytools*), 507
`poly_ring()` (*diofant.domains.domain.Domain* method), 427
`PolyElement` (class in *diofant.polys.rings*), 758
`polygamma` (class in *diofant.functions.special.gamma_functions*), 322
`Polyhedron` (class in *diofant.combinatorics.polyhedron*), 196
`polylog` (class in *diofant.functions.special.zeta_functions*), 365
`PolynomialDivisionFailedError`, 786
`PolynomialError`, 786
`PolynomialRing` (class in *diofant.polys.rings*), 429
`posify()` (in module *diofant.simplify.simplify*), 586
`postorder_traversal()` (in module *diofant.utilities.iterables*), 741
`Pow` (class in *diofant.core.power*), 99
`powdenest()` (in module *diofant.simplify.powsimp*), 597
`power_representation()` (in module *diofant.functions.elementary.complexes*), 278

`fant.solvers.diophantine)`, 628
`powerset()` (`diofant.sets.sets.Set` method), 570
`powsimp()` (`diofant.core.expr.Expr` method), 78
`powsimp()` (in module `diofant.simplify.powsimp`), 596
`pprint()` (in module `diofant.printing.pretty`), 551
`PQa()` (in module `diofant.solvers.diophantine`), 630
`PrecisionExhausted` (class in `diofant.core.evalf`), 138
`preferred_index` (`diofant.functions.special.tensor_functions.KroneckerDelta` property), 392
`prem()` (`diofant.polys.rings.PolyElement` method), 762
`preorder_traversal` (class in `diofant.core.basic`), 54
`pretty()` (in module `diofant.printing.pretty`), 551
`pretty_atom()` (in module `diofant.printing.pretty_symbology`), 563
`pretty_print()` (in module `diofant.printing.pretty`), 551
`pretty_symbol()` (in module `diofant.printing.pretty_symbology`), 563
`prettyForm` (class in `diofant.printing.stringpict`), 564
`PrettyPrinter` (class in `diofant.printing.pretty`), 551
`prev()` (`diofant.combinatorics.prufer.Prufer` method), 200
`prev_binary()` (`diofant.combinatorics.subsets.Subset` method), 204
`prev_gray()` (`diofant.combinatorics.subsets.Subset` method), 204
`prev_lex()` (`diofant.combinatorics.partitions.IntegerPartition` method), 145
`prev_lexicographic()` (`diofant.combinatorics.subsets.Subset` method), 205
`prevprime()` (in module `diofant.ntheory.generate`), 231
`prime()` (in module `diofant.ntheory.generate`), 231
`prime_as_sum_of_two_squares()` (in module `diofant.solvers.diophantine`), 633
`primefactors()` (in module `diofant.ntheory.factor_`), 244
`primepi()` (in module `diofant.ntheory.generate`), 232
`primerange()` (`diofant.ntheory.generate.Sieve` method), 229
`primerange()` (in module `diofant.ntheory.generate`), 232
`primitive()` (`diofant.core.add.Add` method), 106
`primitive()` (`diofant.core.expr.Expr` method), 78
`primitive()` (`diofant.polys.polytools.Poly` method), 521
`primitive()` (`diofant.polys.rings.PolyElement` method), 762
`primitive()` (in module `diofant.polys.polytools`), 533
`primitive_element()` (in module `diofant.polys.numberfields`), 539
`primitive_root()` (in module `diofant.ntheory.residue_ntheory`), 255
`primorial()` (in module `diofant.ntheory.generate`), 233
`principal_branch` (class in `diofant.functions.elementary.complexes`), 279
`print_nonzero()` (`diofant.matrices.matrices.MatrixBase` method), 467
`Printer` (class in `diofant.printing.printer`), 549
`printmethod` (`diofant.printing.ccode.CCodePrinter` attribute), 551
`printmethod` (`diofant.printing.codeprinter.CodePrinter` attribute), 562
`printmethod` (`diofant.printing.fcode.FCodePrinter` attribute), 555
`printmethod` (`diofant.printing.lambdarepr.LambdaPrinter` attribute), 558
`printmethod` (`diofant.printing.latex.LatexPrinter` attribute), 558
`printmethod` (`diofant.printing.mathematica.MCodePrinter` attribute), 557
`printmethod` (`diofant.printing.mathml.MathMLPrinter` attribute), 560
`printmethod` (`diofant.printing.printer.Printer` attribute), 550
`printmethod` (`diofant.printing.repr.ReprPrinter` attribute), 561
`printmethod` (`diofant.printing.str.StrPrinter` attribute), 561
`Product` (class in `diofant.concrete.products`), 265
`product()` (in module `diofant`), 244

fant.concrete.products), 272
ProductSet (class in *diofant.sets.sets*), 576
project() (*diofant.matrices.matrices.MatrixBase* method), 467
Prufer (class in *diofant.combinatorics.prufer*), 199
prufer_rank() (*diofant.combinatorics.prufer.Prufer* method), 200
prufer_repr(*diofant.combinatorics.prufer.Prufer* property), 200
PurePoly (class in *diofant.polys.polytools*), 527
python--m-diofant command line option
 -V, 39
 --auto-symbols, 39
 --help, 39
 --no-ipython, 39
 --no-wrap-division, 39
 --unicode-identifiers, 39
 --version, 39
 --wrap-floats, 39
 -a, 39
 -h, 39
PythonFiniteField (class in *diofant.domains.finitefield*), 431
PythonIntegerRing (class in *diofant.domains.integerring*), 431
PythonRationalField (class in *diofant.domains.rationalfield*), 432

Q

QRdecomposition() (*diofant.matrices.matrices.MatrixBase* method), 443
QRsolve() (*diofant.matrices.matrices.MatrixBase* method), 444
quadratic_residues() (in module *diofant.ntheory.residue_ntheory*), 256
quo() (*diofant.domains.field.Field* method), 428
quo() (*diofant.domains.ring.CommutativeRing* method), 428
quo() (*diofant.polys.polytools.Poly* method), 521
quo() (in module *diofant.polys.polytools*), 534
quo_ground() (*diofant.polys.polytools.Poly* method), 521

R

rad_rationalize() (in module *diofant.simplify.radsimp*), 588
radius_of_convergence (*diofant.functions.special.hyper.hyper* property), 369
radsimp() (*diofant.core.expr.Expr* method), 78
radsimp() (in module *diofant.simplify.radsimp*), 587
randMatrix() (in module *diofant.matrices.dense*), 477
random() (*diofant.combinatorics.perm_groups.PermutationGroup* method), 191
random() (*diofant.combinatorics.permutations.Permutation* class method), 163
random_bitstring() (*diofant.combinatorics.graycode* method), 211
random_complex_number() (in module *diofant.utilities.randtest*), 747
random_integer_partition() (in module *diofant.combinatorics.partitions*), 145
random_poly() (in module *diofant.polys.specialpolys*), 543
random_pr() (*diofant.combinatorics.perm_groups.PermutationGroup* method), 191
random_stab() (*diofant.combinatorics.perm_groups.PermutationGroup* method), 191
randprime() (in module *diofant.ntheory.generate*), 234
Range (class in *diofant.sets.fancysets*), 580
ranges (*diofant.tensor.indexed.Indexed* property), 706
rank (*diofant.combinatorics.graycode.GrayCode* property), 210
rank (*diofant.combinatorics.partitions.Partition* property), 143
rank (*diofant.combinatorics.prufer.Prufer* property), 200
rank (*diofant.tensor.indexed.Indexed* property), 706
rank() (*diofant.combinatorics.permutations.Permutation* method), 163
rank() (*diofant.matrices.matrices.MatrixBase* method), 467
rank_binary (*diofant.combinatorics.subsets.Subset* property), 205
rank_gray (*diofant.combinatorics.subsets.Subset* property), 205
rank_lexicographic (*diofant.combinatorics.subsets.Subset* property), 205
rank_nonlex() (*diofant.combinatorics.permutations.Permutation* property), 205

`method`), 163
`rank_trotterjohnson()` (`diofant.combinatorics.permutations.Permutation` `static method`), 164
`method`), 164
`rat_clear_denoms()` (`diofant.polys.polytools.Poly` `method`), 521
`ratint()` (in module `diofant.integrals.rationaltools`), 402
`ratint_logpart()` (in module `diofant.integrals.rationaltools`), 402
`ratint_ratpart()` (in module `diofant.integrals.rationaltools`), 403
`Rational` (class in `diofant.core.numbers`), 88
`RationalField` (class in `diofant.domains`), 429
`rationalize()` (in module `diofant.parsing.sympy_parser`), 751
`Rationals` (class in `diofant.sets.fancysets`), 579
`ratsimp()` (`diofant.core.expr.Expr` `method`), 78
`ratsimp()` (in module `diofant.simplify.ratsimp`), 593
`rcollect()` (in module `diofant.simplify.ratsimp`), 590
`re` (class in `diofant.functions.elementary.complexes`), 275
`real_root()` (in module `diofant.functions.elementary.miscellaneous`), 302
`real_roots()` (`diofant.polys.polytools.Poly` `method`), 522
`real_roots()` (`diofant.polys.rootoftools.RootOf` `class` `method`), 541
`real_roots()` (in module `diofant.polys.polytools`), 534
`RealAlgebraicField` (class in `diofant.domains`), 429
`RealField` (class in `diofant.domains`), 431
`RealInterval` (class in `diofant.polys.rootisolation`), 784
`Reals` (class in `diofant.sets.fancysets`), 580
`reconstruct()` (in module `diofant.solvers.diophantine`), 634
`recurrence_memo()` (in module `diofant.utilities.memoization`), 747
`red_groebner()` (in module `diofant.polys.groebnertools`), 770
`reduce()` (`diofant.polys.polytools.GroebnerBasis` `method`), 505
`reduce()` (`diofant.sets.sets.Complement` `static method`), 577
`reduce()` (`diofant.sets.sets.Intersection` `static method`), 576
`reduce()` (`diofant.sets.sets.Union` `static method`), 576
`reduce_inequalities()` (in module `diofant.solvers.inequalities`), 610
`reduced()` (in module `diofant.polys.polytools`), 534
`refine()` (`diofant.polys.rootisolation.ComplexInterval` `method`), 784
`refine()` (`diofant.polys.rootisolation.RealInterval` `method`), 785
`refine()` (`diofant.polys.rootoftools.RootOf` `method`), 541
`RefinementFailedError`, 786
`Rel` (in module `diofant.core.relational`), 107
`Relational` (class in `diofant.core.relational`), 108
`rem()` (`diofant.domains.field.Field` `method`), 428
`rem()` (`diofant.domains.ring.CommutativeRing` `method`), 428
`rem()` (`diofant.polys.polytools.Poly` `method`), 522
`rem()` (in module `diofant.polys.polytools`), 534
`remove0()` (`diofant.core.add.Add` `method`), 106
`remove0()` (`diofant.core.expr.Expr` `method`), 78
`render()` (`diofant.printing.stringpict.stringPict` `method`), 564
`reorder()` (`diofant.concrete.expr_with_intlimits.ExprWithIntLimits` `method`), 270
`reorder()` (`diofant.polys.polytools.Poly` `method`), 522
`reorder_limit()` (`diofant.concrete.expr_with_intlimits.ExprWithIntLimits` `method`), 271
`replace()` (`diofant.core.basic.Basic` `method`), 49
`replace()` (`diofant.matrices.matrices.MatrixBase` `method`), 467
`replace()` (`diofant.polys.polytools.Poly` `method`), 522
`reprify()` (`diofant.printing.repr.ReprPrinter` `method`), 561
`ReprPrinter` (class in `diofant.printing.repr`), 561
`reset()` (`diofant.combinatorics.polyhedron.Polyhedron` `method`), 197
`reshape()` (`diofant.matrices.dense.DenseMatrix` `method`), 481

`reshape()` (*diofant.matrices.sparse.SparseMatrixBase* [method](#)), [494](#)
`residue()` (in module *diofant.calculus.residues*), [755](#)
`Result` (class in *diofant.utilities.codegen*), [721](#)
`result_variables` (*diofant.utilities.codegen.Routine* property), [721](#)
`resultant()` (*diofant.polys.polytools.Poly* method), [523](#)
`resultant()` (*diofant.polys.rings.PolyElement* method), [762](#)
`resultant()` (in module *diofant.polys.polytools*), [534](#)
`retract()` (*diofant.polys.polytools.Poly* method), [523](#)
`reverse_order()` (*diofant.concrete.products.Product* method), [266](#)
`reverse_order()` (*diofant.concrete.summations.Sum* method), [264](#)
`reversed` (*diofant.core.relational.Relational* property), [108](#)
`ReversedGradedLexOrder` (class in *diofant.polys.orderings*), [540](#)
`rewrite()` (*diofant.core.basic.Basic* method), [51](#)
`rewrite()` (in module *diofant.calculus.gruntz*), [788](#)
`RGS` (*diofant.combinatorics.partitions.Partition* property), [142](#)
`RGS_enum()` (in module *diofant.combinatorics.partitions*), [146](#)
`RGS_generalized()` (in module *diofant.combinatorics.partitions*), [146](#)
`RGS_rank()` (in module *diofant.combinatorics.partitions*), [147](#)
`RGS_unrank()` (in module *diofant.combinatorics.partitions*), [146](#)
`rhs` (*diofant.core.relational.Relational* property), [109](#)
`right` (*diofant.sets.sets.Interval* property), [574](#)
`right()` (*diofant.printing.stringpict.stringPict* method), [564](#)
`right_open` (*diofant.sets.sets.Interval* property), [574](#)
`ring` (*diofant.domains.ring.CommutativeRing* property), [428](#)
`ring()` (in module *diofant.polys.rings*), [757](#)
`RisingFactorial` (class in *diofant.functions.combinatorial.factorials*), [483](#)
`RL` (*diofant.matrices.sparse.SparseMatrixBase* property), [490](#)
`rmul()` (*diofant.combinatorics.permutations.Permutation* static method), [164](#)
`rmul_with_af()` (*diofant.combinatorics.permutations.Permutation* static method), [165](#)
`root()` (*diofant.polys.polytools.Poly* method), [523](#)
`root()` (in module *diofant.functions.elementary.miscellaneous*), [301](#)
`RootOf` (class in *diofant.polys.rootoftools*), [540](#)
`roots()` (in module *diofant.polys.polyroots*), [542](#)
`RootSum` (class in *diofant.polys.rootoftools*), [541](#)
`Ropen()` (*diofant.sets.sets.Interval* class method), [573](#)
`rot_axis1()` (in module *diofant.matrices.dense*), [478](#)
`rot_axis2()` (in module *diofant.matrices.dense*), [478](#)
`rot_axis3()` (in module *diofant.matrices.dense*), [478](#)
`rotate()` (*diofant.combinatorics.polyhedron.Polyhedron* method), [197](#)
`rotate_left()` (in module *diofant.utilities.iterables*), [742](#)
`rotate_right()` (in module *diofant.utilities.iterables*), [742](#)
`round()` (*diofant.core.expr.Expr* method), [78](#)
`RoundFunction` (class in *diofant.functions.elementary.integers*), [295](#)
`Routine` (class in *diofant.utilities.codegen*), [721](#)
`routine()` (*diofant.utilities.codegen.CodeGen* method), [719](#)
`routine()` (*diofant.utilities.codegen.OctaveCodeGen* method), [721](#)
`row_insert()` (*diofant.matrices.matrices.MatrixBase* method), [468](#)
`row_join()` (*diofant.matrices.matrices.MatrixBase* method), [468](#)
`row_join()` (*diofant.matrices.sparse.MutableSparseMatrix* method), [488](#)
`row_list()` (*diofant.matrices.sparse.SparseMatrixBase* method), [494](#)
`row_op()` (*diofant.matrices.dense.MutableDenseMatrix* method), [483](#)

[row_op\(\)](#) (diofant.matrices.sparse.MutableSparseMatrix method), 489
[row_structure_symbolic_cholesky\(\)](#) (diofant.matrices.sparse.SparseMatrixBase method), 495
[row_swap\(\)](#) (diofant.matrices.dense.MutableDenseMatrix method), 484
[row_swap\(\)](#) (diofant.matrices.sparse.MutableSparseMatrix method), 489
[rref\(\)](#) (diofant.matrices.matrices.MatrixBase method), 468
[rsolve\(\)](#) (in module diofant.solvers.recurr), 684
[rsolve_hyper\(\)](#) (in module diofant.solvers.recurr), 685
[rsolve_poly\(\)](#) (in module diofant.solvers.recurr), 686
[rsolve_ratio\(\)](#) (in module diofant.solvers.recurr), 686
[runs\(\)](#) (diofant.combinatorics.permutations.PermutationGroup method), 165
[runs\(\)](#) (in module diofant.utilities.iterables), 742
S
[S](#) (in module diofant.core.singleton), 55
[s_poly\(\)](#) (in module diofant.polys.groebnertools), 770
[satisfiable\(\)](#) (in module diofant.logic.inference), 426
[scalar_multiply\(\)](#) (diofant.matrices.sparse.SparseMatrixBase method), 495
[schreier_sims\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), 191
[schreier_sims_incremental\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), 191
[schreier_sims_random\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), 192
[schreier_vector\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), 193
[search\(\)](#) (diofant.ntheory.generate.Sieve method), 229
[sec](#) (class in diofant.functions.elementary.trigonometric), 283
[sech](#) (class in diofant.functions.elementary.hyperbolic), 292
[select\(\)](#) (diofant.simplify.epathtools.EPath method), 605
[selections](#) (diofant.combinatorics.graycode.GrayCode property), 210
[separatevars\(\)](#) (in module diofant.simplify.simplify), 582
[series\(\)](#) (diofant.core.expr.Expr method), 771
[Set](#) (class in diofant.sets.sets), 567
[set_domain\(\)](#) (diofant.polys.polytools.Poly method), 523
[set_global_settings\(\)](#) (diofant.printing.printer.Printer class method), 550
[set_modulus\(\)](#) (diofant.polys.polytools.Poly method), 524
[set_order\(\)](#) (diofant.polys.polytools.GroebnerBasis method), 506
[setup\(\)](#) (in module diofant.config), 41
[shape](#) (diofant.matrices.matrices.MatrixBase property), 469
[shape](#) (diofant.tensor.indexed.Indexed property), 706
[shape](#) (diofant.tensor.indexed.IndexedBase property), 707
[ShapeError](#) (class in diofant.matrices.matrices), 473
[Shi](#) (class in diofant.functions.special.error_functions), 348
[shift\(\)](#) (diofant.polys.polytools.Poly method), 524
[Si](#) (class in diofant.functions.special.error_functions), 348
[Sieve](#) (class in diofant.ntheory.generate), 228
[sift\(\)](#) (in module diofant.utilities.iterables), 742
[sig_cmp\(\)](#) (in module diofant.polys.groebnertools), 770
[signature](#) (in module diofant.polys.groebnertools), 770
[sig_mult\(\)](#) (in module diofant.polys.groebnertools), 770
[sign](#) (class in diofant.functions.elementary.complexes), 276
[signature\(\)](#) (diofant.combinatorics.permutations.PermutationGroup method), 165
[signed_permutations\(\)](#) (in module diofant.utilities.iterables), 743
[signinf\(\)](#) (in module diofant.calculus.gruntz), 789
[SimpleDomain](#) (class in diofant.rings.rings), 469

`fant.domains.simplesdomain)`, 428
`simplify()` (`diofant.core.expr.Expr` method), 79
`simplify()` (`diofant.functions.special.delta_functions.DeltaFunctions` method), 318
`simplify()` (`diofant.matrices.dense.MutableDenseMatrix` method), 484
`simplify()` (`diofant.matrices.matrices.MatrixBase` method), 469
`simplify()` (in module `diofant.simplify.simplify`), 580
`simplify_logic()` (in module `diofant.logic.boolalg`), 425
`sin` (class in `diofant.functions.elementary.trigonometric`), 280
`sine_transform()` (in module `diofant.integrals.transforms`), 396
`SineTransform` (class in `diofant.integrals.transforms`), 410
`Singleton` (class in `diofant.core.singleton`), 55
`SingletonRegistry` (class in `diofant.core.singleton`), 56
`SingletonWithManagedProperties` (class in `diofant.core.singleton`), 56
`singular_values()` (`diofant.matrices.matrices.MatrixBase` method), 469
`singularities()` (in module `diofant.calculus.singularities`), 752
`sinh` (class in `diofant.functions.elementary.hyperbolic`), 291
`size` (`diofant.combinatorics.permutations.Permutation` property), 165
`size` (`diofant.combinatorics.polyhedron.Polyhedron` property), 198
`size` (`diofant.combinatorics.prufer.Prufer` property), 201
`size` (`diofant.combinatorics.subsets.Subset` property), 206
`skip()` (`diofant.combinatorics.graycode.GrayCode` method), 210
`skip()` (`diofant.core.basic.preorder_traversal` method), 55
`smoothness()` (in module `diofant.ntheory.factor_`), 244
`smoothness_p()` (in module `diofant.ntheory.factor_`), 245
`solve()` (`diofant.matrices.sparse.SparseMatrixBase` method), 495
`solve()` (in module `diofant.solvers.solvers`), 606
`solve_congruence()` (in module `diofant.ntheory.modular`), 248
`solve_least_squares()` (`diofant.matrices.matrices.MatrixBase` method), 469
`solve_least_squares()` (`diofant.matrices.sparse.SparseMatrixBase` method), 495
`solve_linear()` (in module `diofant.solvers.solvers`), 608
`solve_linear_system()` (in module `diofant.solvers.polysys`), 608
`solve_poly_system()` (in module `diofant.solvers.polysys`), 609
`solve_surd_system()` (in module `diofant.solvers.polysys`), 609
`sort_key()` (`diofant.combinatorics.partitions.Partition` method), 144
`sort_key()` (`diofant.core.basic.Atom` method), 46
`sort_key()` (`diofant.core.basic.Basic` method), 51
`sort_key()` (`diofant.core.expr.Expr` method), 80
`sort_key()` (`diofant.core.numbers.Number` method), 85
`sort_key()` (`diofant.core.symbol.Dummy` method), 82
`SparseMatrix` (in module `diofant.matrices.sparse`), 490
`SparseMatrixBase` (class in `diofant.matrices.sparse`), 490
`spherical_bessel_fn()` (in module `diofant.polys.orthopolys`), 543
`split_super_sub()` (in module `diofant.printing.conventions`), 562
`split_symbols()` (in module `diofant.parsing.sympy_parser`), 750
`split_symbols_custom()` (in module `diofant.parsing.sympy_parser`), 750
`spoly()` (in module `diofant.polys.groebnertools`), 771
`sqf()` (in module `diofant.polys.polytools`), 535
`sqf_list()` (`diofant.polys.polytools.Poly` method), 524
`sqf_list()` (in module `diofant.polys.polytools`), 535
`sqf_norm()` (`diofant.polys.polytools.Poly` method), 524
`sqf_norm()` (in module `diofant.polys.polytools`), 535
`sqf_normal()` (in module `diofant.solvers.diophantine`), 634

`sqf_part()` (*diofant.polys.polytools.Poly* method), 524
`sqf_part()` (in module *diofant.polys.polytools*), 535
`sqrt()` (in module *diofant.functions.elementary.miscellaneous*), 302
`sqrt_mod()` (in module *diofant.ntheory.residue_ntheory*), 256
`sqrt_mod_iter()` (in module *diofant.ntheory.residue_ntheory*), 256
`sqrtdenest()` (in module *diofant.simplify.sqrtdenest*), 599
`square_factor()` (in module *diofant.ntheory.factor_*), 245
`square_factor()` (in module *diofant.solvers.diophantine*), 633
`srepr()` (in module *diofant.printing.repr*), 561
`sstr()` (in module *diofant.printing.str*), 561
`stabilizer()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 193
`stack()` (*diofant.printing.stringpict.stringPict* static method), 564
`standard_transformations` (in module *diofant.parsing.sympy_parser*), 750
`start` (*diofant.sets.sets.Interval* property), 574
`StdFactKB` (class in *diofant.core.assumptions*), 44
`stirling()` (in module *diofant.functions.combinatorial.numbers*), 316
`StrictGreaterThan` (class in *diofant.core.relational*), 115
`StrictLessThan` (class in *diofant.core.relational*), 118
`stringify_expr()` (in module *diofant.parsing.sympy_parser*), 749
`stringPict` (class in *diofant.printing.stringpict*), 563
`strong_gens` (*diofant.combinatorics.perm_groups.PermutationGroup* property), 194
`StrPrinter` (class in *diofant.printing.str*), 561
`sturm()` (*diofant.polys.univar.UnivarPolyElement* method), 764
`subfactorial` (class in *diofant.functions.combinatorial.factorials*), 310
`subgroup_search()` (*diofant.combinatorics.perm_groups.PermutationGroup* method), 194
`subresultants()` (*diofant.polys.polytools.Poly* method), 525
`subresultants()` (*diofant.polys.rings.PolyElement* method), 762
`subresultants()` (in module *diofant.polys.polytools*), 535
`Subs` (class in *diofant.core.function*), 128
`subs()` (*diofant.core.basic.Basic* method), 52
`subs()` (*diofant.matrices.immutable.ImmutableSparseMatrix* method), 497
`subs()` (*diofant.matrices.matrices.MatrixBase* method), 470
`Subset` (class in *diofant.combinatorics.subsets*), 202
`subset` (*diofant.combinatorics.subsets.Subset* property), 206
`subset_from_bitlist()` (*diofant.combinatorics.subsets.Subset* class method), 206
`subset_from_indices()` (*diofant.combinatorics.subsets.Subset* class method), 206
`subsets()` (in module *diofant.utilities.iterables*), 743
`Sum` (class in *diofant.concrete.summations*), 262
`sum_of_four_squares()` (in module *diofant.solvers.diophantine*), 627
`sum_of_powers()` (in module *diofant.solvers.diophantine*), 628
`sum_of_squares()` (in module *diofant.solvers.diophantine*), 628
`sum_of_three_squares()` (in module *diofant.solvers.diophantine*), 627
`summation()` (in module *diofant.concrete.summations*), 271
`sup` (*diofant.sets.sets.Interval* property), 574
`sup` (*diofant.sets.sets.Set* property), 571
`superset` (*diofant.combinatorics.subsets.Subset* property), 207
`superset_size` (*diofant.combinatorics.subsets.Subset* property), 207
`support()` (*diofant.combinatorics.permutations.Permutation* method), 166
`swinnerton_dyer_poly()` (in module *diofant.polys.specialpolys*), 543
`Symbol` (class in *diofant.core.symbol*), 80
`symbols()` (in module *diofant.core.symbol*), 82
`syntactic_group()` (*diofant.combinatorics.generators* method), 169

`symmetric_difference()` (diofant.sets.sets.Set method), 571
`symmetric_poly()` (in module diofant.polys.specialpolys), 543
`symmetric_residue()` (in module diofant.ntheory.modular), 249
`SymmetricGroup()` (in module diofant.combinatorics.named_groups), 212
`symmetrize()` (in module diofant.polys.polyfuncs), 538
`sympify()` (in module diofant.core.sympify), 41
`sysode_linear_neq_order1()` (in module diofant.solvers.ode), 677

T

`T` (diofant.matrices.expressions.MatrixExpr property), 500
`T` (diofant.matrices.matrices.MatrixBase property), 444
`table()` (diofant.matrices.matrices.MatrixBase method), 470
`tail_degree()` (diofant.polys.rings.PolyElement method), 763
`tan` (class in diofant.functions.elementary.trigonometric), 282
`tanh` (class in diofant.functions.elementary.hyperbolic), 292
`taylor_term()` (diofant.core.expr.Expr method), 80
`taylor_term()` (diofant.functions.elementary.hyperbolic.csch static method), 293
`taylor_term()` (diofant.functions.elementary.hyperbolic.sinh static method), 291
`TC()` (diofant.polys.polytools.Poly method), 508
`tensorcontraction()` (in module diofant.tensor.array), 701
`tensorproduct()` (in module diofant.tensor.array), 701
`terminal_width()` (diofant.printing.stringpict.stringPict method), 564
`terms()` (diofant.polys.polytools.Poly method), 525
`terms_gcd()` (diofant.polys.polytools.Poly method), 525
`terms_gcd()` (in module diofant.polys.polytools), 536
`termwise()` (diofant.polys.polytools.Poly method), 525
`to_cnf()` (in module diofant.logic.boolalg), 423
`to_dnf()` (in module diofant.logic.boolalg), 423
`to_exact()` (diofant.polys.polytools.Poly method), 525
`to_expr()` (diofant.domains.domain.Domain method), 427
`to_field()` (diofant.polys.polytools.Poly method), 526
`to_nnf()` (in module diofant.logic.boolalg), 424
`to_prufer()` (diofant.combinatorics.prufer.Prufer static method), 201
`to_rational()` (diofant.domains.RealField method), 431
`to_ring()` (diofant.polys.polytools.Poly method), 526
`to_tree()` (diofant.combinatorics.prufer.Prufer static method), 201
`together()` (diofant.core.expr.Expr method), 80
`together()` (in module diofant.polys.rationaltools), 544
`tolist()` (diofant.matrices.dense.DenseMatrix method), 481
`tolist()` (diofant.matrices.sparse.SparseMatrixBase method), 496
`total_degree()` (diofant.polys.polytools.Poly method), 526
`total_degree()` (diofant.polys.rings.PolyElement method), 763
`totient` (class in diofant.ntheory.factor_), 246
`Trace` (class in diofant.matrices.expressions), 502
`trace()` (diofant.matrices.matrices.MatrixBase method), 471
`trager()` (in module diofant.polys.factorization_alg_field), 775
`trailing()` (in module diofant.ntheory.factor_), 246
`transform()` (diofant.integrals.integrals.Integral method), 407
`transform_variable` (diofant.integrals.transforms.IntegralTransform property), 409
`transformation_to_DN()` (in module dio-

`fant.solvers.diophantine)`, 621
`transformation_to_normal()` (in module `diofant.solvers.diophantine`), 635
`transitivity_degree` (diofant.combinatorics.perm_groups.PermutationGroup property), 195
`transpose` (class in diofant.functions.elementary.complexes), 280
`Transpose` (class in diofant.matrices.expressions), 502
`transpose()` (diofant.core.expr.Expr method), 80
`transpose()` (diofant.matrices.expressions.BlockedDiagonalMatrix method), 504
`transpose()` (diofant.matrices.matrices.MatrixBase method), 471
`transpositions()` (diofant.combinatorics.permutations.PermutationGroup method), 166
`tree_cse()` (in module `diofant.simplify.cse_main`), 603
`tree_repr` (diofant.combinatorics.prufer.Prufer property), 202
`trial_division()` (in module `diofant.polys.modulargcd`), 783
`trigamma()` (in module `diofant.functions.special.gamma_functions`), 325
`trigintegrate()` (in module `diofant.integrals.trigonometry`), 404
`trigsimp()` (diofant.core.expr.Expr method), 80
`trigsimp()` (in module `diofant.simplify.trigsimp`), 593
`trunc()` (diofant.polys.polytools.Poly method), 526
`trunc()` (in module `diofant.polys.polytools`), 537
`Tuple` (class in diofant.core.containers), 139
`tuple_count()` (diofant.core.containers.Tuple method), 139

U

`U()` (in module `diofant.printing.pretty_symbolology`), 562
`ufuncify()` (in module `diofant.utilities.autowrap`), 714
`UfuncifyCodeWrapper` (class in diofant.utilities.autowrap), 713
`Unequality` (class in diofant.core.relational), 115
`unflatten()` (in module `diofant.utilities.iterables`), 744
`unicode_identifiers()` (in module `diofant.interactive.session`), 567
`UnicodeError`, 786
`unify()` (diofant.domains.domain.Domain method), 427
`unify()` (diofant.polys.polytools.Poly method), 526
`Union` (class in diofant.sets.sets), 575
`union()` (diofant.sets.sets.Set method), 571
`uniq()` (in module `diofant.utilities.iterables`), 744
`UnivariatePolynomialError`, 786
`UnivarPolyElement` (class in diofant.polys.univar), 763
`UnivarPolynomialRing` (class in diofant.polys.univar), 430
`UnivariateLeadingCoefficientError`, 786
`unrad()` (in module `diofant.simplify.sqrtdenest`), 600
`unrank()` (diofant.combinatorics.graycode.GrayCode class method), 211
`unrank()` (diofant.combinatorics.prufer.Prufer class method), 202
`unrank_binary()` (diofant.combinatorics.subsets.Subset class method), 207
`unrank_gray()` (diofant.combinatorics.subsets.Subset class method), 207
`unrank_lex()` (diofant.combinatorics.permutations.Permutation class method), 166
`unrank_nonlex()` (diofant.combinatorics.permutations.Permutation class method), 166
`unrank_trotterjohnson()` (diofant.combinatorics.permutations.Permutation class method), 167
`upper` (diofant.tensor.indexed.Idx property), 705
`upper_triangular_solve()` (diofant.matrices.matrices.MatrixBase method), 471
`uppergamma` (class in diofant.functions.special.gamma_functions), 325
`use()` (in module `diofant.simplify.traversaltools`), 604

V

`values()` (diofant.core.containers.Dict method), 140

`values()` (`diofant.matrices.matrices.MatrixBase` method), 471
`var()` (in module `diofant.core.symbol`), 84
`variables` (`diofant.concrete.expr_with_limits.ExprWithLimits` property), 268
`variables` (`diofant.core.function.Derivative` property), 125
`variables` (`diofant.core.function.Lambda` property), 121
`variables` (`diofant.core.function.Subs` property), 129
`variables` (`diofant.utilities.codegen.Routine` property), 721
`vec()` (`diofant.matrices.matrices.MatrixBase` method), 471
`vech()` (`diofant.matrices.matrices.MatrixBase` method), 472
`vectorize` (class in `diofant.core.multidimensional`), 120
`verify_derivative_numerically()` (in module `diofant.utilities.randtest`), 747
`verify_numerically()` (in module `diofant.utilities.randtest`), 748
`vertices` (`diofant.combinatorics.polyhedron.Polyhedron` property), 198
`viete()` (in module `diofant.polys.polyfuncs`), 538
`vobj()` (in module `diofant.printing.pretty_symbology`), 563
`vstack()` (`diofant.matrices.matrices.MatrixBase` class method), 472

W
`width()` (`diofant.printing.stringpict.stringPict` method), 564
`Wild` (class in `diofant.core.symbol`), 81
`WildFunction` (class in `diofant.core.function`), 122
`wrap_float_literals()` (in module `diofant.interactive.session`), 567
`write()` (`diofant.utilities.codegen.CodeGen` method), 719
`wronskian()` (in module `diofant.matrices.dense`), 476

X
`xobj()` (in module `diofant.printing.pretty_symbology`), 562
`Xor` (class in `diofant.logic.boolalg`), 420
`xreplace()` (`diofant.core.basic.Basic` method), 53

`xreplace()` (`diofant.matrices.immutable.ImmutableSparseMatrix` method), 497
`xreplace()` (`diofant.matrices.matrices.MatrixBase` method), 472
`xsym()` (in module `diofant.printing.pretty_symbology`), 563

Y
`yn` (class in `diofant.functions.special.bessel`), 355
`Ynm` (class in `diofant.functions.special.spherical_harmonics`), 385
`Ynm_c()` (in module `diofant.functions.special.spherical_harmonics`), 387

Z
`Zero` (class in `diofant.core.numbers`), 92
`ZeroMatrix` (class in `diofant.matrices.expressions`), 503
`zeros()` (`diofant.matrices.dense.DenseMatrix` class method), 482
`zeros()` (`diofant.matrices.sparse.SparseMatrixBase` class method), 496
`zeros()` (in module `diofant.matrices.dense`), 473
`zeta` (class in `diofant.functions.special.zeta_functions`), 363
`zip_row_op()` (`diofant.matrices.dense.MutableDenseMatrix` method), 484
`zip_row_op()` (`diofant.matrices.sparse.MutableSparseMatrix` method), 489
`Znm` (class in `diofant.functions.special.spherical_harmonics`), 387